TYMSHARE TYMCOM-X MANUALS
REFERENCE SERIES

ADDENDUM TO
FORTRAN IV

FEBRUARY 1973

TYMSHARE, INC.

CUPERTINO, CALIFORNIA 95014

TYMSHARE®

CONTENTS

Page

Page

SECTION 1

INTRODUCTION

The TYMCOM-X FORTRAN is constantly being improved through new features and expanded capability. This document describes the Tymshare FORTRAN modifications and is intended as a supplement to the FORTRAN capability described in the Tymshare TYMCOM-X FORTRAN IV Reference Manual.

The major improvements presented in this supplement are Tymshare's file handling capability, program segmentation, and the FORTRAN program development package, CFORTRAN. The file handling capability is described in Section 2, along with other new input/output features. Segmentation is the subject of Section 3, and CFORTRAN is the subject of Section 4.

Section 5 describes the new and modified library subroutines which are available for use in FORTRAN programs. In conjunction with the program transfer capability of the RUN subroutine, the /ENTRY statement permits multiple entry point specification in the called program.

The final section, Section 6, discusses the binary library program, CARMEL, which permits both the creation and the modification of subprogram libraries.

The Appendix is a list of the subroutines in the Scientific Subroutine Package mentioned on page 57.

In all examples in this document, everything typed by the user is underlined. The symbol used to indicate a user-typed Carriage Return is ↵.

Control characters are denoted by a superscript c. For example, A^C denotes Control A. The method for typing a control character depends on the type of terminal used. Consult the literature for your particular terminal or see your Tymshare representative.

Lowercase letters used in examples of command forms represent the input to be typed. In the command

>SAVE file name↵

the characters file name indicate that the user should type a file name in that position.

Brackets indicate an option; they are not part of the statement or command. For example,

>LIST [TO file name]↵

indicates that the user may optionally specify an output file.

SECTION 2

INPUT AND OUTPUT

The major improvement in FORTRAN input and output procedures is the Tymshare file handling capability, which permits all file processing to be accomplished with simple, definitive statements. Also described in this section is the M format specification for inserting and reading commas and decimal points embedded in a number. The last item discussed is asterisk output to indicate that a number is too large for the specified FORMAT field.

FORTRAN FILE HANDLING

The FORTRAN file handling capability is designed for maximum efficiency and flexibility, while maintaining the traditionally simple syntax of Tymshare's software. TYMCOM-X FORTRAN allows as many as 15 files to be open simultaneously. FORTRAN files may contain symbolic or binary code, and the user may select sequential or random processing.

To provide the user with improved price performance, the TYMCOM-X FORTRAN system for file handling incorporates major improvements in information storage allocation methods. All file information is packed to utilize fully the storage devices. Most existing FORTRAN files are compatible with the file handling methods described below. A simple program for converting incompatible binary files is described on page 11.

File Descriptions

A variety of terms describes files, each term referring to a particular aspect of the file. The terms symbolic and binary refer to the code in which the data in the file is written; sequential and random refer to the modes of accessing the file information.

A symbolic file contains alphanumeric characters and can be printed in TYMEX or read and displayed in EDITOR or CFORTRAN;[1] symbolic file input and output must be formatted. A binary file contains binary code and generally requires less storage space than a symbolic file; a binary file cannot be listed in TYMEX or CFORTRAN or read into EDITOR; all binary file input and output must be unformatted. A symbolic file information unit may be a character or a record. A binary file information unit is a 36-bit word or a record. One line of a symbolic file is a series of characters terminated by a Carriage Return and a Line Feed. A record is a user-defined unit of information containing a specified number of characters for a symbolic file or a specified number of words for a binary file.

1 - CFORTRAN is discussed on page 33.

The terms sequential and random refer to the manner in which the file information is accessed. Sequential file information is read or written in the sequence in which the data is stored on the file. A random mode permits any unit of information to be accessed without processing any other part of the file. Generally, sequential files require less program overhead than random files, but may be considerably slower for input and output of only selected information.

Some of the file handling statements permit the user to specify a file position, thereby selecting the information to be processed in a random mode. A file position is described in characters, words, or records, and refers to the next information unit to be processed. Position 1 refers to the first information unit on the file. A position has meaning only for random files; sequential file information is always processed sequentially. A file may be written sequentially and read randomly, or written randomly and read sequentially.

The OPEN Statement

Before any activity occurs on a file within a FORTRAN program, the file is opened with the OPEN statement. The OPEN statement activates the file, and the statement arguments provide all the necessary information to describe the file. The form of the OPEN statement is

OPEN(file number,'file name',mode(n),data type,ERR=label)

where the arguments are as defined below.

file number — An integer constant or variable corresponding to the number of the file being opened. This number is used to identify the file in subsequent file handling statements. If the terminal is used for input and output, 14 files may be open simultaneously; if the terminal is not used for any input or output, 15 files may be open simultaneously. Permissible file numbers are 1 through 29; each open file must have a unique file number.

'file name' — The name of the file being opened; the file name must start with a letter from A to Z and may include letters and the digits 0 through 9. This argument may be the actual file name enclosed in single quote marks or the name of a variable, array, or array element. The file name may include a preceding user name enclosed in parentheses and a six-character file name plus a period (.) and a three-character file name extension. The user must specify any applicable file name extension, as none is assumed in file handling;

blanks or extra characters in a file name are ignored. In addition, the file name may be preceded by a device name and a colon (:); the file name is then optional. For example,

```
OPEN(18,'TTY:',OUTPUT)
```

opens the terminal as unit 18 for symbolic output. When no device is named, FORTRAN assumes DSK.

mode(n) — The access mode defining a READ or WRITE operation and sequential or random processing. For random access files, the user may define a record length by specifying the n enclosed in parentheses. The record length is an integer constant equal to the number of characters per record for a symbolic file or the number of words for a binary file. The character count for a symbolic record must include the two terminating characters: a Carriage Return and a Line Feed. The user may omit the access mode entirely, and the system assumes the INPUT mode. The five access modes are defined as follows:

INPUT	Sequential input — read from file.
OUTPUT	Sequential output — write on file. If an existing file is opened in the OUTPUT mode, the previous contents are erased and the new information written when the file is closed. Sequential files are not written if the user interrupts program execution.
RANDIN.	Random input — read information from selected positions on the file.
RANDOUT	Random output — write information at selected positions on the file.
RANDIO	Random input and output — read or write at specified positions of the file.

data type — Specifies the file code and may be either SYMBOLIC or BINARY. If the data type is omitted, the system assumes SYMBOLIC.

ERR=label — Specifies the label of the statement to which control is transferred if an error occurs in the opening operation. This argument is optional and, if included, functions as a FORTRAN GO TO statement; if it is omitted, an error causes the system to print an error message and terminate program execution.

Examples

OPEN(4,'ABC.DAT')

Opens file ABC.DAT as file number 4. FORTRAN assumes that the file is symbolic and that the access mode is sequential input.

OPEN(7,'FILEA',RANDIO(5),BINARY,ERR=20)

Opens FILEA as file number 7 for random input and output of records containing five words each. FILEA is a binary file, and the user wants to transfer control to statement number 20 if an error occurs in the opening process.

If the user omits the OPEN statement, a subsequent READ or WRITE statement implies the file name FORn.DAT, where n is a two-digit file number. If the READ or WRITE statement does not contain a FORMAT statement reference, FORTRAN assumes that FORn.DAT is a binary file. Also, unless explicitly opened with an OPEN statement, file number 5 is interpreted as the terminal.

The CLOSE Statement

Each file used in a FORTRAN program may be closed with the CLOSE statement. The form of the CLOSE statement is

CLOSE(file number)

where the file number is specified in the same manner as described for the OPEN statement. For example,

CLOSE(15)

closes file number 15.

If a file has not been closed with a CLOSE statement, it is automatically closed at the end of program execution. If the user interrupts a running program, any file open for input is automatically closed; if a file is open for sequential output, nothing is written on the file.

The READ and WRITE Statements

The READ statement permits information to be read from a file and values to be assigned to variables within the FORTRAN program. The form of the READ statement is:

READ(file number#position,format label,ERR=label,END=label)input list

The WRITE statement permits information in a FORTRAN program to be written on a file. The form of the WRITE statement is:

WRITE(file number#position,format label,ERR=label)output list

The arguments in the READ and WRITE statements are identical in form and function, except that READ includes the additional argument END. The optional END argument specifies the label of the statement to which control transfers when an end of file is encountered during input. If the user omits this argument and the system encounters an end of file, FORTRAN prints an error message and terminates program execution. The file number argument and the ERR argument are defined for the OPEN statement on page 4. The other arguments are described below.

position	An expression truncated to an integer value specifying the file position associated with the next random input or output operation. As discussed on page 4, this argument is significant only for random files; if it is specified for sequential file input and output, it produces an error.
format label	The label of the FORMAT statement, or the name of an array containing the format, to be used for the input or output operation. This argument is required for symbolic input and output, but is not valid for binary files since all binary input and output must be unformatted.
input/output list	Standard FORTRAN input/output list of variables or arrays used in the data transfer.

Examples

READ(4)X,Y,Z,U,V,W

Reads a value for each variable in the input list from file number 4. Because the READ statement contains no FORMAT reference, file number 4 must be binary.

READ(7#2*J+K,35,ERR=50,END=40)ARRAY

Reads values for ARRAY, starting from position 2*J+K on file number 7 according to FORMAT statement 35. If an error occurs, FORTRAN transfers control to statement 50; if FORTRAN encounters the end of the file, it transfers control to statement 40.

WRITE(6)((ARRAY(I,J),I=1,3)J=1,5)

Writes the array elements indicated on file number 6, starting at the current file position. This WRITE statement contains no FORMAT reference; file number 6 must be binary.

WRITE(17#57,20,ERR=50)

Writes according to FORMAT statement 20 at position 57 of file number 17. Because a position argument appears in a WRITE statement, file number 17 must be a random file opened for output or for input and output. A FORMAT reference is legal for symbolic files only; therefore, file number 17 must be a symbolic file.

Symbolic Files

A READ or WRITE statement referring to a symbolic file must contain a FORMAT statement reference and may contain an input/output list.

A line consists of a series of characters terminated by a Carriage Return and a Line Feed. During input, the end of a FORMAT statement causes FORTRAN to seek a Carriage Return and a Line Feed as the READ terminator. For example, the file XYZ contains the lines

```
1234.56789AAC
7965.3477.8RTF
9773.996RMT
```

where each line is terminated by a Carriage Return and a Line Feed, and the user's program contains the statements:

```
      LOGICAL L
      OPEN(3,'XYZ')
      READ(3,9)N1,A1,B1,R1
9     FORMAT(I2,F5.2,F3.1,A3)
      READ(3,9)N2,A2,B2,R2
      READ(3,10)X,Y,Z,L
10    FORMAT(I3,4X,I1,A2,L1)
```

All three lines are read, and the variable values assigned are:

N1=12	A1=34.56	B1=78.9	R1=AAC
N2=79	A2=65.34	B2=77.0	R2=8RT
X=977	Y=6	Z=RM	L=T (true)

Note that a READ statement format seeks a Carriage Return and a Line Feed when the end of the FORMAT statement is encountered, and information which has not been read is skipped. If the file being read does not contain Carriage Returns and Line Feeds, FORTRAN skips all remaining information, encountering an end of file as the result of the search for a Carriage Return and a Line Feed.

In a WRITE statement referring to a file for which the user has not defined a record, a single FORMAT statement causes FORTRAN to write a Carriage Return and a Line Feed when the end of the FORMAT statement is encountered. For example, the statements

```
        X=2.345
        Y=6.973
        ALPHA=16.491
        BETA=10.393
        OPEN(4,'DATA',OUTPUT)
        WRITE(4,20)X,Y,ALPHA,BETA
        WRITE(4,30)X,Y,ALPHA,BETA
20      FORMAT(2F7.3)
30      FORMAT(2F9.4)
```

cause FORTRAN to write

```
  2.345  6.973
 16.491 10.393
   2.3450   6.9730
  16.4910  10.3930
```

on file number 4. Each time the end of a FORMAT statement is reached, FORTRAN writes a Carriage Return and a Line Feed. The Carriage Return and Line Feed terminators can be suppressed by including a dollar sign ($) as the last field specification in the FORMAT statement.

If the user defines a record length for the file being processed, all symbolic file READ and WRITE statements must include a FORMAT statement label. In READ statements for fixed length random files, a referenced FORMAT statement containing fewer character specifications than the number of characters defined for the record skips the remaining characters in the record. A FORMAT statement containing more characters than one record causes FORTRAN to continue reading successive records until the format is satisfied, then scan to the end of the current record. A WRITE statement FORMAT reference specifying fewer characters than the defined record length causes FORTRAN to insert a blank for each missing character. A WRITE statement FORMAT reference specifying more characters than the defined record length writes as many records as required, inserts any necessary blanks, and terminates the current record with a Carriage Return and a Line Feed.

A symbolic file READ statement without an input/output list causes FORTRAN to read according to the specified format, but the values are not stored since no variable storage locations are specified.

Binary Files

No FORMAT statement may be used for a binary READ or WRITE operation, because all binary file operations must be unformatted. If no input/output list appears in a READ statement for a binary file, the system skips one record if the record length is defined; if a record length is not defined, no activity occurs. A binary READ or WRITE operation processes the number of words required by the input/output list. When a record length is defined, if the input/output list requires less than one record, FORTRAN scans to the end of the record; if the input/output list requires more than one record, FORTRAN reads successive records, then scans to the end of the current record.

The POSITION Statement

As an alternate method of setting a file position, the user may use the POSITION statement rather than specify the position argument in the READ or WRITE statement. The position designated in the POSITION statement pertains to the next input or output operation. The general statement form is

POSITION(file number,position)

where the file number is specified in the manner described for the OPEN statement on page 4, and the position may be an expression evaluated as an integer. For example,

POSITION(5,32)

sets the current position on file number 5 to 32.

The POSITION and SIZE Functions

FORTRAN includes two integer functions which return information about open random files. The value of the POSITION function is the current file position measured in characters, words, or records; the value of the SIZE function is the total number of characters, words, or records on the file. The form of the POSITION function is:

POSITION(file number)

The form of the SIZE function is:

SIZE(file number)

Example

The user wants to write 15 successive values from the array at every fifth position on the file, starting from but not including the current position. The user does not want to write beyond the last position in the file. He uses both the SIZE function and the POSITION function to write his random file.

```
      DIMENSION SALES(150)
      OPEN(15,'PARTNO',RANDOUT,BINARY)
      S=SIZE(15)
      DO 10 J=1,15
      WRITE(15#(POSITION(15)+5,ERR=50) SALES(J)
      IF(POSITION(15)+5.GE.S) GO TO 30
10    CONTINUE
```

Simulated Terminal Output

The user may instruct FORTRAN to write data on a file as though the terminal was being used for output. The statement is:

CALL SIMTTY(file number)

SIMTTY causes the carriage control character to be suppressed during file output.

Binary File Conversion

FORTRAN binary files other than those created in the manner described in this document must be converted. To convert a binary file, the user calls BINCON from TYMEX and enters the name of the file he wants to convert. For example,

```
-R BINCON↵
FILE TO CONVERT: HISTRY↵
EXIT
-
```

converts the file HISTRY to a new binary file form. This program should be used only once for each file.

Sample Problem

The user has a file containing employee number, pay rate, and hours worked. His program reads the data from this file, permits him to update any information, and modifies the input file. The program then computes the gross pay for each employee and writes the information on a second file.

```
-TYPE PRF↵                    The user lists his input file.
 99   2.00 48.00
 77   2.50 40.00
 88   3.75 40.00
 55   5.50 40.00
 66   3.35 40.00
 44   2.10 48.00

-TYPE PAYROL↵                 He then lists his program file.
        DIMENSION EMP(100),RATE(100),HRS(100),PAY(100)
        OPEN(3,'PRF',RANDIO,SYMBOLIC)
 7      FORMAT(I2)
        READ(3#1,8,END=33)(EMP(I),RATE(I),HRS(I),I=1,20)
 33     N=I-1
 35     TYPE 2
 2      FORMAT(' ENTER ANY UPDATES'/)
        ACCEPT 8,EMPNO,RT,HR
        IF(EMPNO.EQ.0)GO TO 25
        DO 20 I=1,100
        IREC=I
        IF(EMPNO.EQ.EMP(I))GO TO 30
 20     CONTINUE
 25     CLOSE(3)
        OPEN(4,'GPF',OUTPUT,SYMBOLIC)
        DO 200 I=1,N
        J=EMP(I)
        PAY(J)=HRS(I)*RATE(I)
        WRITE(4,5,ERR=210)EMP(I),PAY(J)
 200    CONTINUE
 5      FORMAT(I3,2X,F7.2)
 210    CLOSE(4)
        STOP
 30     POSITION(3,1+(IREC-1)*17)
        RATE(IREC)=RT
        HRS(IREC)=HR
        WRITE(3,8)EMPNO,RT,HR
        GO TO 35
 8      FORMAT(I3,2F6.2)
        END

-EXECUTE PAYROL↵              The user executes his program.
F-IV:  PAYROL
 MAIN.
LOADING
EXECUTION
```

```
ENTER ANY UPDATES
 77 2.50 48.↵

ENTER ANY UPDATES
55 6.00 40.↵

EXIT

-TYPE PRF↵
 99  2.00 48.00
 77  2.50 48.00
 88  3.75 40.00
 55  5.50 40.00
 66  3.35 40.00
 44  2.10 48.00

-TYPE GPF↵
 99    96.00
 77   120.00
 88   150.00
 55   220.00
 66   134.00
 44   100.80
```

He enters the changes to the information on the input file.

The user lists the updated input file, then the output file.

M FORMAT SPECIFICATION

The M format specification permits the reading and writing of integer variables with embedded commas and a specified number of decimal places. The M format specification works only with FORTRAN integers. If the user enters a number with an M format specification, FORTRAN converts the digits to an integer value. If the user requests output with an M format specification, FORTRAN causes a decimal point to be printed in the position the user specifies and a comma to be printed between every three digits to the left of the decimal point.

The general form of the M format specification is

M w.d

where w is the total width of the field, and d is the number of digits to appear to the right of the decimal point.

Because the M format conversions work only with integer variables, the magnitude of the numbers may not exceed the maximum value of FORTRAN integers, 34, 359, 738, 367.

Output

When the M format specification is used for output, if the w is not specified, or is equal to 0, a field width of 15 is assumed; if the d is not specified, or is equal to 0, no decimal point is printed. The last position in the field is reserved for a sign. If the value printed is negative, a minus sign is printed in this position. If the value printed is positive, the last position is blank. Sufficient space must be allocated in the specification or the number will be truncated from the left. For example,

```
        I=123456
        TYPE 10,I
10      FORMAT(1X,M4)
```

results in the output of only four columns as follows:

456b

where b represents a blank in the position reserved for a sign.

The following are examples of integers printed using an M format specification:

Format	Integer	Output
M10	12345	bbb12,345b
M10.0	12345	bbb12,345b
M10.2	12345	bbb123.45b
M10.2	-12345	bbb123.45-

Input

Any number entered with the M format specification is automatically converted to a FORTRAN integer. Provided the specified field length is adequate, the integer stored consists of all the digits to the left of the decimal point plus the specified number of digits to the right of the decimal point. The number entered may include commas and blanks; both are counted as part of the field width, but do not affect the value of the number. For example, if the format specification

FORMAT(M8.2)

is used for input, the first eight characters, including commas and blanks, are read. When more than two digits appear to the right of the decimal point, extra digits are ignored, that is, truncated. If fewer than two digits appear to the right of the decimal point, or are not read because the field width is too small, a zero is read for each missing digit. A sign may appear in either the first or last position in the field; the field width, 8, includes the sign position.

The following are examples of input using M format with a specified field width:

Format	Input	Stored Integer
M8.2	12,345,678.90	12345600
M8.2	-12345678.90	-123456700
M8.2	1234567.89	123456700
M8.2	123456.789	12345670
M8.2	-123456.789	-12345600
M8.2	12,345.67-	-1234560
M8.2	123456.78-	12345670

Either w or d may be zero in the specification. If d is zero, no decimal places are read. If the field width, w, is zero, the input may have any number of digits and a sign. The value limit of a FORTRAN integer must not be exceeded. A series of numbers entered according to an M format specification of field width zero must be separated by any non-digit character other than blank, comma, decimal point, plus sign, or minus sign. For example, the following numbers are entered with an M format of field width zero:

Format	Input	Stored Integer
M0.2	1,234, 56 7.89-	-123456789
M0.0	12345. 6789 3	12345
M0.5	12345.678912	1234567891

OUTPUT ERRORS

Occasionally, an output value contains more digits than the specified format allows. In this case, the entire format field is filled with asterisks. For example, if the value of K is 7594 and the program contains the statements

```
        TYPE 1000,K
1000    FORMAT(' THE NUMBER IS',I3)
```

the resulting output is:

```
THE NUMBER IS ***
```

SECTION 3

PROGRAM SEGMENTATION

Segmentation provides maximum flexibility in program construction to minimize core storage requirements. By segmenting a program and defining which segments reside in core at successive times during program execution, it is possible to use all or part of core repeatedly during different portions or stages of the program. Data may be saved during all or part of execution in blank or labeled COMMON declared and allocated according to the rules for segmentation. This capability results in direct cost savings and permits FORTRAN IV programs of almost unlimited size.

PROGRAM COMPONENTS

Each program segment consists of a main program and/or subprograms written on a single file. Each segment is created separately and may be compiled separately. For example, the user writes four files: M1, S1, S2, and S3. M1 contains the main program MAIN., subroutines SUBR1, SUBR2, and SUBR3, and the function FUNCT1. S1 contains five subroutines; S2 contains three subroutines and four functions; and S3 contains four subroutines.

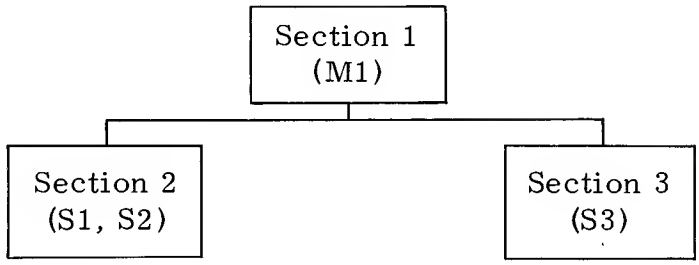
Segments grouped together such that they are in core simultaneously and can be replaced only as a unit form a program section. The section which contains the main program is called the main program section. For example, the segments M1, S1, S2, and S3 may comprise three sections as follows:

Section 1	Section 2	Section 3
M1	S1 and S2	S3

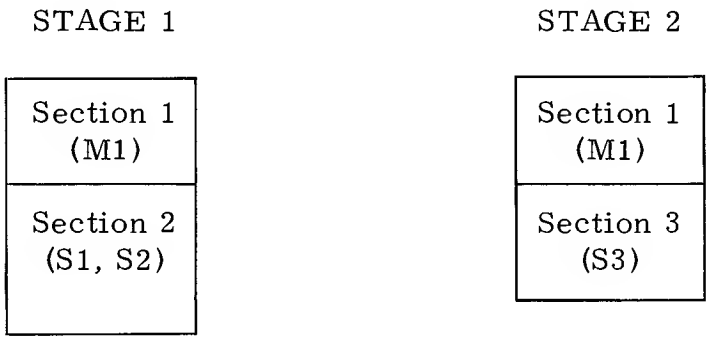
Segments S1 and S2 grouped in this way comprise one section, since they must be in core simultaneously and must be replaced as a unit. Because segment M1 contains the main program, Section 1 is the main program section.

Program sections are associated in levels defined by their starting location in storage. A program section and the section which replaces it at the same starting location in core are of the same level. For example, the user creates the files M1, S1, S2, and S3 and groups them in three sections such that Section 1 must be in core throughout execution, but when Section 2 is in core with Section 1, Section 3 is not needed, and when Section 3 is in core, Section 2 is not needed. The user may structure his

program such that Section 3 replaces Section 2 in core when needed. Diagrammatically his program is:



In the first stage of the program the user needs Section 1 and Section 2 in core. In stage 2, the user needs Section 1 and Section 3 in core. The following illustrates core during successive stages of the program.



Section 3 replaces Section 2; both sections have the same starting location in core. Section 2 and Section 3 are of the same level.

The actual core used is the core required by the loaded sections. If a smaller section replaces a larger section, only the core required for the smaller section is used, ensuring minimum cost to the user. For example, assume Section 1 requires 10K core, Section 2 requires 8K core, and Section 3 requires 4K core. During stage 1 operations, the program uses 18K core, but when Section 3 replaces Section 2, the program uses only 14K core.

PROGRAM STRUCTURE

If the main program section of the program remains in core throughout execution, and additional sections are replaced for successive operations, the program is said to have an overlay structure. If the main program section is replaced during execution, the program has a link structure. Each main program section of a link structure may have an overlay structure associated with it.

The number of segments in core at one time depends on the user's definition of the program structure. The maximum number of sections is 256; the number of levels is otherwise unlimited.

The storage arrangement of the segments and the replacement scheme are normally specified in the LOAD command. The files named in the LOAD command may be either symbolic or relocatable binary.

The Overlay Structure

The user defines an overlay structure with the LOAD command, and the system automatically stores the structure on an overlay file and saves the file in the user's directory. Besides the overlay file, the LOAD command creates a core image which normally contains the main program section of the overlay structure; the user may save this core image with the SAVE command. The EXECUTE command or FDEBUG command may be used in place of the LOAD command, but the core image can not be saved.

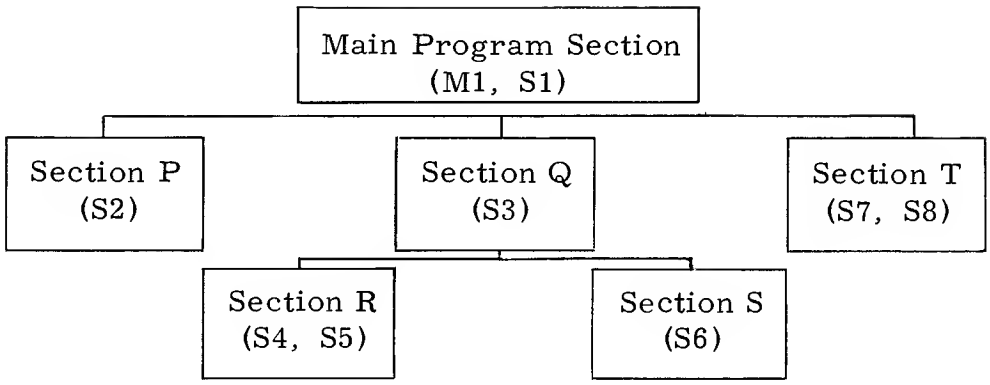
Assume that segments of a program have been created separately and stored on files M1, S1, S2, S3, S4, S5, S6, S7, and S8, and that the user wishes to execute the program in stages with the following segment combinations:

STAGE 1	M1, S1, and S2
STAGE 2	M1, S1, S3, S4, and S5
STAGE 3	M1, S1, S3, and S6
STAGE 4	M1, S1, S7, and S8

The program can be structured such that:

1. M1 and S1 are loaded and remain in core during the entire execution.
2. S2 is called into core when needed and used for stage 1 only.
3. S3, S4, and S5 replace S2 for stage 2 operations.
4. S6 replaces S4 and S5 for stage 3 operations.
5. S7 and S8 replace S3 and S6 for stage 4 and remain in core for the remainder of execution.

Diagramed, the structure is:



The user creates this structure with the following LOAD command:

-LOAD M1,S1,[S2!S3,[S4,S5!S6]!S7,S8]↵

A comma separates files which are to be contiguous in core; an exclamation mark separates files which are to overlay each other; brackets define the boundaries of successive levels. Note that the two levels below the main program section in the diagram correspond to the two sets of brackets in the LOAD command. The innermost set of brackets defines the lowest level of the program structure. The meaning of the comma and exclamation mark can also be related to the diagram. The third level of the structure is defined by:

[S4,S5!S6]

The comma indicates that S4 and S5 are to be contiguous in core. The exclamation mark indicates that S4 and S5 are to be replaced by S6. There are two level 3 sections. One section contains segments S4 and S5; the other section contains segment S6.

If a segment is to appear as all or part of more than one section, it must be specified in each position. For example, if segment S7 is to reside in core as part of Section P as well as part of Section T, the LOAD command is:

-LOAD M1,S1,[S2,S7!S3,[S4,S5!S6]!S7,S8]↵

This structure is permitted only if M1 and S1 do not contain a FORTRAN call to any subroutine in S7, because the subroutine name is a doubly-defined symbol and the system does not know which segment to load.

Note that two separate copies of S7 are created when S7 is specified twice. Thus, S7 in Section P is completely independent of S7 in Section T, and any variable changes in S7 in one section do not affect the S7 variables in the other section.

File access information is retained during the entire execution of an overlay structure for access in every segment of the program. This reduces the time required for file input and output, and is accomplished with a dynamic buffer allocation scheme which uses core only as required.

If the user's program consists of several segments, the LOAD command may be rather lengthy. The user may type the information on successive lines by ending the line to be continued with a semicolon and a Carriage Return.

The user may eliminate any need to retype the LOAD command by writing a command string file containing the information for the LOAD command. He uses this file by typing an @ and the file name following the word LOAD. For example, the user creates a command string file in EDITOR and uses that file in the LOAD command.

```
-EDITOR↵
*APPEND↵
M1,S1,[S2!S3,;↵
[S4,S5!S6]!S7,S8]↵
*WRITE SPROG1↵
 NEW FILE↵
35 CHRS
*QUIT↵

-LOAD @SPROG1↵
```

A line to be continued must be terminated by a semicolon preceding the Carriage Return. The content of the command string file is identical to the information the user types in the LOAD command when not using a command string file.

Overlay Files

The program structure is saved on an overlay file created automatically by the LOAD command. The LOAD command also creates a core image which may be explicitly saved with the SAVE command.

Normally, the overlay file assumes the name of the first file specified in the LOAD command and contains the file name extension .OVL. The user may elect a different file name. He names the file by inserting /NAME and a file name between the word LOAD and the structure definition or by typing /NAME and a file name after the structure definition. Thus, if the user wishes to name his file ABPROG.NEW, he types:

```
-LOAD /NAME ABPROG.NEW M1,S1,[S2!S3,[S4,S5!S6]!S7,S8]↵
```

or

```
-LOAD M1,S1,[S2!S3,[S4,S5!S6]!S7,S8]/NAME ABPROG.NEW↵
```

As in the above examples, he may specify the file name extension; if no file name extension is given, the file has the file name extension .OVL.

The user saves the core image on a file by typing

-SAVE file name↵

where the file name may be any name the user chooses and may include a file name extension. If no file name extension is specified, the core image file has the file name extension .SAV. The user may change the name of the core image file at any time, but the name of the overlay file is stored on the core image file and must remain the same for the program to be run without reloading.

The core image contains only the main program section unless the user types a /INITIAL in the LOAD command anywhere within the last section he wishes to have stored on the core image; each higher level connecting section is automatically saved. The section containing the /INITIAL and each higher level connecting section must be the last sections named for their respective overlay levels. For example, in the structure defined by the command

-LOAD M1,S1,[S2!S3,[S4,S5!S6]!S7,S8]↵

and depicted in the diagram on page 20, the only section which may be saved with the main program section is Section T containing S7 and S8, since Section T is the last section named in level 2. Because successively lower levels are loaded contiguously in core and no lower level section is associated with Section T, no lower level section can be saved in the core image.

To run a segmented program without reloading, both the overlay file and the core image file must exist in the same user directory. The user executes his program with the RUN command as follows:

-RUN file name↵

The user does not need to specify the file name extension if it is .SAV, because the RUN command assumes a core image file with the extension .SAV. If the files are in another user's directory, the command is:

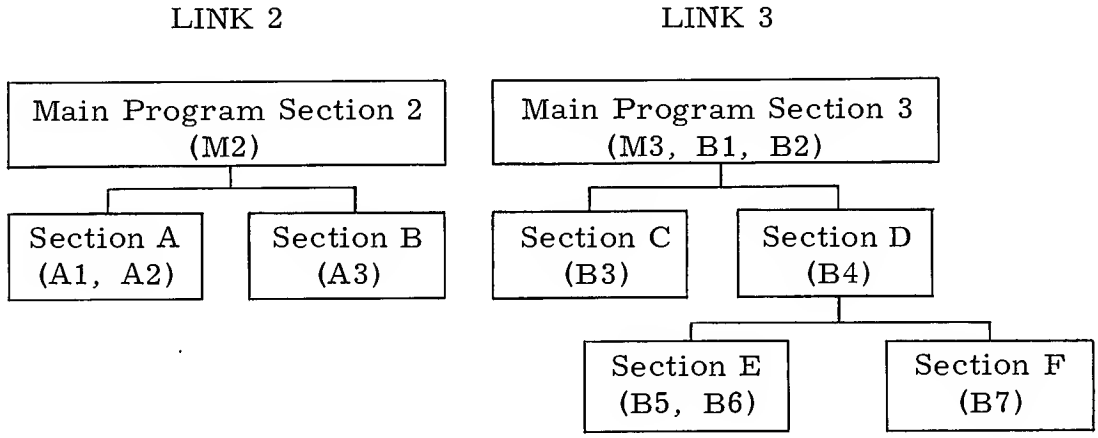
-RUN (user name)file name↵

The overlay file normally remains open throughout execution and counts as one of the 15 files available for use in FORTRAN IV programs. The user may type /CLOSE anywhere in the LOAD command containing the structure definition and the system automatically opens the overlay file when a new section is called into core, and recloses the file when the required sections are loaded. This open and close process is repeated as often as necessary during program execution, freeing the file for temporary use.

The Link Structure

The main program section of a program may be replaced during execution, and each main program section may have an overlay structure associated with it. Only one main program may be in core at any one time.

The form of the LOAD command defining a link structure is merely an extension of the form used to define an overlay structure. For example, the user wishes to create a program with three links. The first link is the main program section and overlay structure described on page 20. The second and third links are illustrated below.



The segments have been created separately and exist on separate files.

The LOAD command is:

-LOAD [M2,[A1,A2!A3]!M3,B1,B2,[B3!B4,[B5,B6!B7]];↵
!M1,S1,[S2!S3,[S4,S5!S6]!S7,S8]]↵

One chain file is created and named M2.OVL because M2 is the first segment named in the LOAD command. The user may specify the file name with the /NAME file name convention discussed for overlay files. A core image exists for the main program section of the last link defined, that is, the section containing M1 and S1. This core image must be explicitly saved for execution with the RUN command as follows:

-SAVE file name↵

-RUN file name↵

The order of the link specifications in the LOAD command is unimportant except that a core image exists only for the last link defined. Because execution starts from the main program section on the core image file, the first link to be executed must be specified last in the LOAD command.

The /INITIAL and /CLOSE capabilities discussed for overlay files also apply to link structure definitions. [1] In addition, the command string file may be used in the LOAD command defining a link structure. [2]

CALLING SEQUENCE

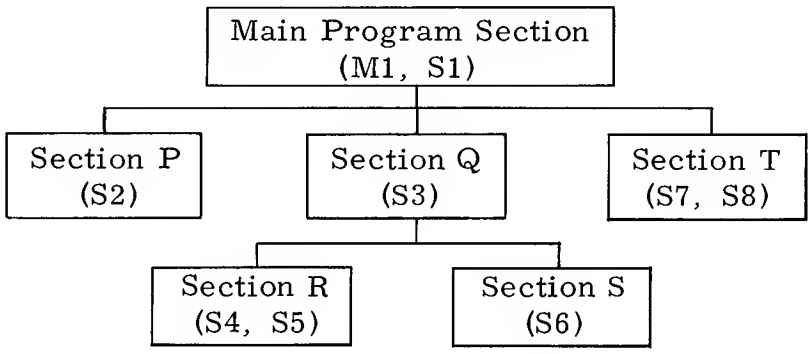
Functionally, links are merely an extension of overlays. The primary difference is the manner in which sections are called into core. Within an overlay structure, the loading order of sections is controlled by FORTRAN CALL statements to subprograms. Successive links are called into core by a CALL CHAIN statement in the current link.

NOTE: Each time a section is read, the previous contents of that core area are destroyed.

Overlay Calls

Normally, the core image file named in the RUN command contains only the main program section, and only this section is initially loaded. The order in which other sections are loaded is determined by the order in which FORTRAN CALL statements are encountered in the program. A legal FORTRAN CALL statement to a subprogram causes the section containing that subprogram to be loaded. This is true for all normal FORTRAN subroutine and function calls except those in which the subprogram being called is passed to the caller as an argument.

A diagram of the overlay structure defined on page 19 is shown below.



The segments comprising each section are given in parentheses.

1 - See page 21 for a discussion of /NAME, /INITIAL, and /CLOSE.

2 - See page 21 for a description of the command string file.

If the section containing the called subprogram is one level below the main program section, only that section is loaded. If the section called is farther down in the structure, the connecting sections at each higher level are also loaded; that is, using the above diagram, if a subprogram in Section S is called, Section Q is also loaded. If a section is not in core, its subprograms can be called only from a higher level and the calling section must be the main program section or a loaded connector section between the main program section and the called section. In the example above, if Section R is not in core, it may be called from the main program section or from Section Q, provided Section Q is already in core. These are the only possible ways Section R may be called into core. Further, Section P and Section T may not call any other section to be loaded. Any legal call to a section subprogram causes the entire section to be loaded. For example, with the above program structure, a call to any subprogram in S7 or S8 causes all of Section T to be loaded.

Any subprogram call is permitted if the section containing the subprogram called and the section containing the call are both in core when the CALL statement is encountered. For example, if Section Q and Section S are in core with the main program section, Section S may contain a FORTRAN CALL to a subprogram in Section Q.

NOTE: Any subprogram call which requires replacement of the calling subprogram is not permitted.

FORTRAN statements may call a section any number of times, but each time it is called, the section's position in core is governed by the original structure definition.

Link Calls

At the end of execution of a link, the next link is called into core with the FORTRAN statement

CALL CHAIN('file name')

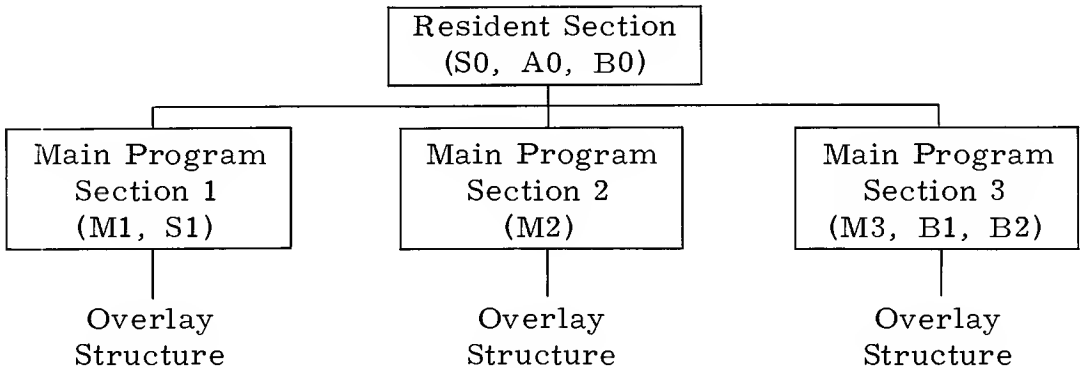
where the file name corresponds to the file name of the first segment specified in the link. The file name conventions discussed on page 58 of this document apply, except that no file name extension may be specified in the CALL CHAIN statement. For example,

CALL CHAIN('M3')

causes the link whose first segment is M3 to be loaded in core.

RESIDENT SEGMENTS

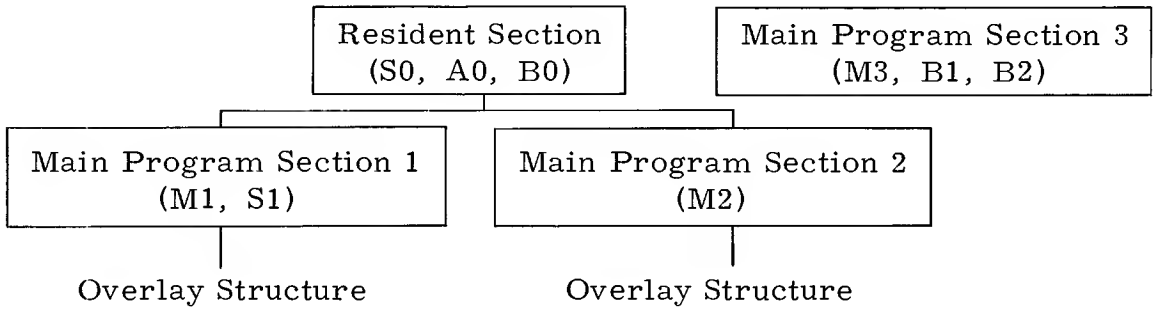
One or more segments may be resident in core throughout execution even though the program has a link structure, causing the main program section to be replaced. For example, the user wishes to retain the subprogram segments S0, A0, and B0 in a resident section while three main program sections and their overlay structures are executed. In diagram form, the structure is shown below.



The LOAD command form is:

-LOAD S0,A0,B0,[M2,[overlay structure]!M3,B1,B2,[overlay structure];↵
!M1,S1,[overlay structure]]↵

A resident section may be shared by some of the links, replacing or being replaced by successive links as in the following example structure:



The main program sections in this example are of different levels and have different starting locations in core. Main Program Section 1 and Main Program Section 2 are still of the same level and share the Resident Section. The LOAD command to create this structure is:

-LOAD [M3,B1,B2,[overlay structure]!S0,A0,B0,[M2,[overlay structure];↵
!M1,S1,[overlay structure]]]↵

Because of the order given in the LOAD command, the saved core image file contains the Resident Section and Main Program Section 1. The saved core image file always contains a main program section and any higher-level sections. Additional sections may be saved by using /INITIAL.[1]

The CALL CHAIN statement may appear in any section, regardless of the number of sections sharing a given higher-level section. This is true even if the main program sections are of different levels.

NOTE: If the program has an overlay structure, that is, only one main program, the resident section is also the main program section.

DATA RETENTION

COMMON storage is allocated the first time it is declared. It may be declared in any section, but the section in which COMMON is first declared must be a section which can call all segments sharing the data in COMMON. For example, assume the same program structure shown in the diagram on page 24.

Case 1 — Section R and Section S are to share data in COMMON. Only Section Q and the Main Program Section can call both Section R and Section S. The COMMON must, therefore, be declared in Section Q or the Main Program Section.

Case 2 — COMMON is to be shared by Section P and Section S. Section S can be called from Section Q or the Main Program Section, but Section P can be called from the Main Program Section only. The shared COMMON must be declared in the Main Program Section.

Data to be used in more than one link must appear in blank or labeled COMMON declared in a resident section which does not contain a main program. The resident section must be in core with each link sharing the data in COMMON. For example, in the first program structure diagram in the above section, page 26, the blank or labeled COMMON declared in the resident section is retained for use throughout execution. In the second program structure diagram in the above section, page 26, the COMMON data declared in the resident section is retained for use in the links associated with Main Program Section 1 and 2 only. No data is retained for use in the link associated with Main Program Section 3. Because the link associated with Main Program Section 3 shares no data and no subprograms with the rest of the structure, it might just as easily be executed as a separate program.

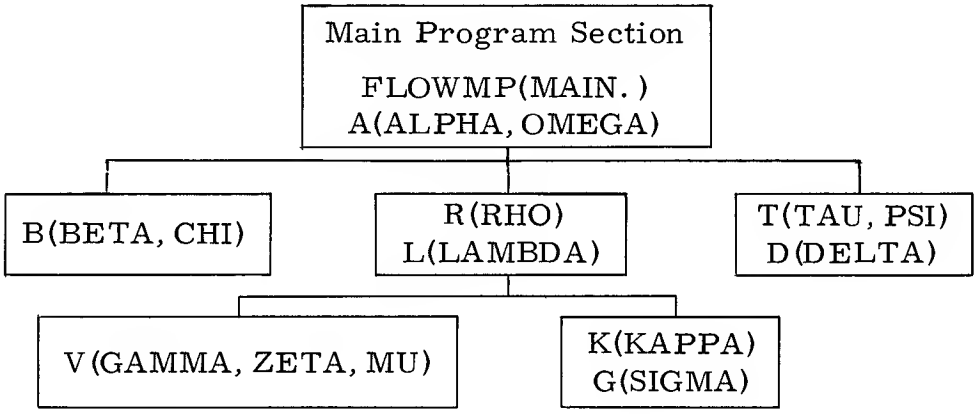
1 - For a discussion of /INITIAL, see page 22.

LIBRARY LOADING

The loader performs a library search immediately following the loading of each individual segment and loads any library program called in that segment. Since the loader checks only for undefined symbols, a user subprogram with the same name as a library routine which is referred to prior to its loading causes the library routine of the same name to be loaded, and all future references to that name result in a multiply-defined symbol error.

SAMPLE PROBLEM

This sample problem demonstrates program segmentation by printing messages in the main program and each subprogram. The user creates his program in segments and saves the segments on separate files; he designs his program with the following structure.



The user loads his program, saves the core image on the file FLOW, and begins execution of the program with the RUN command. The messages from the main program and each subprogram are separated by two blank lines.

```
-LOAD FLOWMP,A,[B!R,L,[V!K,G]!T,D]↵
F-IV:  FLOWMP
 MAIN.
F-IV:  A
 ALPHA
 OMEGA
F-IV:  B
 BETA
 CHI
F-IV:  R
 RHO
F-IV:  L
 LAMBDA
F-IV:  V
 GAMMA
 ZETA
 MU
F-IV:  K
 KAPPA
F-IV:  G
 SIGMA
F-IV:  T
 TAU
 PSI
F-IV:  D
 DELTA
LOADING
7K CORE
```

The user specifies the program structure, and the system automatically creates the overlay file and names it FLOWMP.OVL.

The files named in the LOAD command may be either symbolic or relocatable binary. The files named here are symbolic and each file is compiled separately.

```
-SAVE FLOW↵
JOB SAVED
```

The user saves the core image on the file FLOW.SAV. The file name extension is automatically assigned by the system.

```
-RUN FLOW↵
```

The user initiates program execution from the core image file. The .SAV file name extension is implied.

```
THIS IS THE MAIN PROGRAM.
ALPHA AND OMEGA ARE ALSO IN THE MAIN PROGRAM SECTION.
THE FIRST SUBROUTINE CALLED IS ALPHA.


CONTROL IS NOW IN ALPHA.
ALPHA CONTAINS A CALL TO OMEGA.
ANY NUMBER OF CALCULATIONS MAY BE PERFORMED IN THE MAIN PROGRAM SECTION
BEFORE ANY OTHER SECTION IS LOADED.


CONTROL IS IN OMEGA WHICH WAS CALLED FROM ALPHA.
OMEGA CALLS BETA CAUSING THE LEVEL 2 SECTION CONTAINING BETA AND CHI TO
BE LOADED.


CONTROL HAS BEEN TRANSFERRED TO BETA AND BETA CALLS CHI.
```

CHI CONTINUES OPERATIONS FOR THAT LEVEL 2 SECTION AND RETURNS CONTROL TO BETA.

BETA RETURNS CONTROL TO OMEGA.

OMEGA RETURNS CONTROL TO ALPHA.

ALPHA CALLS GAMMA.

CONTROL IS IN GAMMA, A THIRD LEVEL SECTION CALLED FROM THE MAIN PROGRAM SECTION.
THE LEVEL 2 SECTION CONNECTING THE MAIN PROGRAM SECTION AND THE CALLED THIRD LEVEL SECTION IS ALSO LOADED.

GAMMA CALLS ZETA.

ZETA RETURNS CONTROL TO GAMMA.

GAMMA CALLS MU.

MU RETURNS CONTROL TO GAMMA.

GAMMA RETURNS CONTROL TO ALPHA.

ALPHA RETURNS CONTROL TO THE MAIN PROGRAM.

RHO, LAMBDA, ZETA, GAMMA, AND MU ARE CURRENTLY IN CORE.
THE MAIN PROGRAM CALLS RHO.

RHO CALLS LAMBDA.

CONTROL IS IN LAMBDA WHICH CALLS SIGMA. THE LEVEL 3 SECTION CONTAINING KAPPA AND SIGMA IS LOADED.

CONTROL IS IN SIGMA AND KAPPA IS CALLED.

KAPPA RETURNS CONTROL TO SIGMA.

SIGMA RETURNS CONTROL TO LAMBDA.

LAMBDA RETURNS CONTROL TO RHO.

RHO RETURNS CONTROL TO THE MAIN PROGRAM.

THE MAIN PROGRAM CALLS DELTA AND THE LEVEL 2 SECTION CONTAINING TAU, PSI, AND DELTA IS LOADED.

DELTA CALLS PSI.

PSI CALLS TAU.

TAU RETURNS CONTROL TO PSI.

PSI RETURNS CONTROL TO DELTA.

DELTA RETURNS CONTROL TO THE MAIN PROGRAM.
EXIT

SECTION 4

CFORTRAN COMMANDS

CFORTRAN is a program development package which combines a debugging and editing capability with the TYMCOM-X FORTRAN IV language. The user may create, edit, run, and debug his program by entering commands and statements directly in CFORTRAN. CFORTRAN contains commands which are not part of the programming language, but are special commands allowing easy, efficient program manipulation. This section describes these special CFORTRAN commands.

The user accesses CFORTRAN by typing

-CFORTRAN↵

at TYMEX command level. A greater than sign (>) prints on the terminal to indicate that the user may enter CFORTRAN commands or statements.

CFORTRAN requires fixed format program statements; that is, character positions 1 through 6 are reserved for statement numbers, comment indicators, and line continuation characters. The actual FORTRAN statement must be within character positions 7 through 72. Character position 1 immediately follows a CFORTRAN line number.[1] A user-typed Control I moves the carriage to the appropriate position for entering a FORTRAN statement. For example:

>10 50I^C CONTINUE↵

Each time the user enters a command specifying an output file, CFORTRAN responds with the message NEW FILE or OLD FILE. A user-typed Carriage Return or Line Feed confirms the command; any other character aborts the command. If the user types a Carriage Return in response to the OLD FILE message, CFORTRAN writes the new information over the previous contents of the file.

Each command may be abbreviated to as few characters as required to identify it uniquely. For example, the MOVE command may be abbreviated to MOV, but no further, to distinguish it from the MODIFY command.

SAMPLE PROBLEM

File name arguments may be established by an EDIT, SAVE, COMPILE, or RUN command, then omitted in subsequent uses of these commands. CFORTRAN, when reasonable, assumes the previously established file name argument. The sample problem on the following page illustrates the implied file name argument feature, the simple CFORTRAN editing capabilities, and the direct entry of FORTRAN statements.

1 - CFORTRAN line numbers are discussed on page 36.

```
-CFORTRAN ↵

>EDIT THETA ↵          The user enters his program from the file THETA.F4.
OK                     CFORTRAN anticipates a .F4 file name extension.
>LIST ↵

  5          REAL M1,M2
 10          DIMENSION X(4),Y(4)
 18          TYPE 5
 19 5        FORMAT(' ENTER TWO POINTS FOR EACH OF TWO LINES'/)
 20          DO 10 I=1,4
 30 10       ACCEPT 20,X(I),Y(I)
 39 C COMPUTE THE SLOPE OF EACH LINE
 40          M1=(Y(1)-Y(2))/(X(1)-X(2))
 50          M2=(Y(3)-Y(4))/(X(3)-X(4))
 59 C COMPUTE THE ANGLE BETWEEN THE TWO LINES
 60          THETA=ATAN((M1-M2)/(1+M1*M2))
 64 20       FORMAT(2F8.3)
 65          TYPE 30,M1,M2
 66 30       FORMAT(' SLOPES = ',F8.3,2X,' AND ',F8.3/)
 70          TYPE 31,THETA
 80 31       FORMAT(' ANGLE BETWEEN LINES = ',F10.6,' RAD.')
100          END

>MOVE 66 TO 79 ↵       The user modifies his program by moving statements from one position
>MOVE 64 TO 78 ↵       to another, renumbering the lines, and changing a FORMAT statement.
>RENUMBER AS 10(10) ↵
>MODIFY $-1 ↵
160 31       FORMAT(' ANGLE BETWEEN LINES = ',F10.6,' RAD.'//)-
>SAVE ↵                He saves the modified program on THETA.F4.
THETA.F4
OLD FILE ↵             The Carriage Return verifies the intent to write on THETA.F4.
OK
>COMPILE ↵             The user checks the compilation of his program. CFORTRAN
F-IV:  THETA.F4        saves the compiled program on THETA.REL.
 MAIN.

>RUN ↵                 The user executes his program. CFORTRAN assumes the same
LOADING                file name argument originally established by the EDIT command.

DEBUG!
*START ↵

ENTER TWO POINTS FOR EACH OF TWO LINES
2.5 3.4 ↵              The user enters the X and Y coordinates for each point.
-3. 5.2 ↵
2. -2. ↵
4. 2. ↵

SLOPES =    -0.327    AND     2.000

ANGLE BETWEEN LINES =  -1.423435 RAD.


>QUIT ↵

-
```

PROGRAM LISTINGS

With the LIST and FAST commands, the user may instruct CFORTRAN to list all or part of his program on the terminal or to a file. The LIST command lists all or part of a program with line numbers vertically aligned, and has the general form:

>LIST [line address list] [TO file name]↵

The FAST command lists statements on the terminal exactly as entered by the user, or on a file in a special format. The general form of the FAST command is:

>FAST [line address list] [TO file name]↵

Line address lists are described on page 36.

The commands

>LIST↵

and

>FAST↵

instruct CFORTRAN to list the entire program on the terminal.

To list all or part of a program on a file, the user includes the TO file name option. The command

>LIST TO file name↵

lists the entire program on the file in the same format as:

>LIST↵

The command

>FAST TO file name↵

lists the entire program on the specified file in the same form as the SAVE command described on page 43. The file name is exactly as the user enters it; no file name extension is automatically assigned.

The following example illustrates the listings produced by the LIST and FAST commands:

```
>LIST↵

  .75             ACCEPT 20,A,B
 2      20        FORMAT(2F8.3)
 3.001            C=SQRT(A**2+B**2)
 4                TYPE 1,A,B,C
50.05   1         FORMAT(3F8.3)
60                CALL BSUB
82.5              END
```

```
>FAST↵
.75          ACCEPT 20,A,B
2 20       FORMAT(2F8.3)
3.001          C=SQRT(A**2+B**2)
4          TYPE 1,A,B,C
50.05 1        FORMAT(3F8.3)
60          CALL BSUB
82.5          END
>
```

LINE NUMBERS AND ADDRESSES

Each FORTRAN program statement must have a line number between .001 and 99999.999; CFORTRAN commands must be entered without line numbers. Line numbers are independent of statement labels; a FORTRAN control statement refers to the statement label. For example, in the line

35 10 CONTINUE

35 is the line number, and 10 is the FORTRAN statement label. Thus,

DO 10 I=1,N

is a legal reference to the above line, but

DO 35 I=1,N

is not.

CFORTRAN permits various methods of line addressing to specify a single line or range of lines. The address of a range of lines always takes the form

$a_1 : a_2$

where a_1 is the address of the first line in the range, and a_2 is the address of the last line in the range. A line address list includes a series of line addresses, separated by commas; embedded blanks are not permitted.

Line addresses may take any of the following forms:

Address	Refers To
n	Line number n
*n	The n^{th} line in the program
.	The last line operated on
$	The last line in the program
a+n	The n^{th} line after line a
a-n	The n^{th} line before line a

Example

To illustrate the various methods of line addressing, assume that the following CFORTRAN program is used:

```
>LIST↵

 2          DATA A,B/2.7,7.1/
 5          ACCEPT 1,C
 9 1        FORMAT(F8.3)
17          D=A*C+B
21          TYPE 1,D
25          END

>FAST 9↵        This command addresses line number 9.
9 1        FORMAT(F8.3)
>FAST *4↵       The *4 indicates the fourth line in the program.
17          D=A*C+B
>LIST *3:*5↵

 9 1        FORMAT(F8.3)
17          D=A*C+B
21          TYPE 1,D

>LIST .↵        The user wants to list the last line operated on.

21          TYPE 1,D

>FAST $↵            The $ refers to the last line in the program.
25          END
>LIST 9:$↵

 9 1        FORMAT(F8.3)
17          D=A*C+B
21          TYPE 1,D
25          END

>LIST 2+2:$-1↵

 9 1        FORMAT(F8.3)
17          D=A*C+B
21          TYPE 1,D

>
```

The command

```
>FAST .,2:21 TO FILE2↵
```

writes the current line and lines 2 through 21 on file FILE2.

ENTERING STATEMENTS FROM THE TERMINAL

CFORTRAN permits the user to enter program statements from the terminal with any one of three forms. The user may enter a single line including the line number; he may specify a range of line numbers, then enter the lines without numbers; or he may specify a starting line number with a line number increment, and CFORTRAN prompts for each line by printing the appropriate line number. In all cases, the word ENTER is optional.

To enter a single line from the terminal, the general form is:

>[ENTER] line number statement↵

If the line number specified in the command is the line number of a statement in the program, the line typed replaces the original line. If the line number specified is not in the program, the new line is inserted into the program according to line number order. For example:

```
>LIST↵

1          DIMENSION X(10),Y(10),Z(10),W(10)
2          DO 10 I=1,10
3          Z(I)=X(I)+Y(I)
4          W(I)=(X(I)+Y(I))**2
5  5       CONTINUE
6          END

>4          W(I)=Z(I)**2↵
>4.1          TYPE 20,X(I),Y(I),Z(I),W(I)↵
>4.2 20       FORMAT(4F12.4)↵
>5  10      CONTINUE↵
>LIST↵

1            DIMENSION X(10),Y(10),Z(10),W(10)
2            DO 10 I=1,10
3            Z(I)=X(I)+Y(I)
4            W(I)=Z(I)**2
4.1          TYPE 20,X(I),Y(I),Z(I),W(I)
4.2 20       FORMAT(4F12.4)
5   10       CONTINUE
6            END

>
```

New lines 4 and 5 replace the old lines 4 and 5. Lines 4.1 and 4.2 are inserted in the appropriate numerical order.

A line or a range of lines may be entered without typing line numbers by using the following form:

>[ENTER] line number:line number↵

CFORTRAN prompts with a @ at the beginning of each line, and the user may type his lines without line numbers. To terminate the command, the user types a Carriage Return immediately after the @ prompt. CFORTRAN assigns line numbers to the lines typed, beginning with the first number in the range and choosing as an increment the first of 1, .1, .01, and .001 that will allow the lines typed to fit in the specified range. CFORTRAN deletes all statements with line numbers in the specified range, then inserts the new lines. For example:

```
>LIST↵
1           DIMENSION X(10),Y(10),Z(10),W(10)
2           DO 10 I=1,10
3           Z(I)=X(I)+Y(I)
4           W(I)=(X(I)+Y(I))**2
5           CONTINUE
6           END

>ENTER 4:5↵
@           W(I)=Z(I)**2↵
@           TYPE 20,X(I),Y(I),Z(I),W(I)↵
@20         FORMAT(4F12.4)↵
@10         CONTINUE↵
@↵
>LIST↵

1             DIMENSION X(10),Y(10),Z(10),W(10)
2             DO 10 I=1,10
3             Z(I)=X(I)+Y(I)
4             W(I)=Z(I)**2
4.1           TYPE 20,X(I),Y(I),Z(I),W(I)
4.2 20        FORMAT(4F12.4)
4.3 10        CONTINUE
6             END

>
```

To specify a starting line number, an increment, and, optionally, a terminating line number, the command form is:

>[ENTER] line number(increment)[line number]↵

When the optional terminating line number is specified, CFORTRAN first deletes all lines from the starting line number through the terminating line number, then prompts for each line by printing the appropriate line number. The command is terminated when the terminating line number is reached or when the user types a Carriage Return following a line number

prompt. In the following example, CFORTRAN first deletes lines 10 through 100, then prompts for successive lines until the user types a Carriage Return.

```
>ENTER 10(5)100↵
10          A=5.↵
15          B=A**2↵
20          X=SQRT(A+B)↵
25          Y=SQRT(B-A)↵
30 ↵
>
```

If the terminating line number is omitted, statements are accepted until the user types a Carriage Return immediately after a line number prompt. Thus, the above statements could also be entered as follows without deleting lines 30 through 100.

```
>ENTER 10(5)↵
10          A=5.↵
15          B=A**2↵
20          X=SQRT(A+B)↵
25          Y=SQRT(B-A)↵
30 ↵
>
```

When using this form, the user is protected against accidental changing or intermixing of program lines. The user may enter the beginning line number even if it currently exists, but if CFORTRAN computes a line number which would cause changing or intermixing of lines, CFORTRAN automatically terminates the command instead of giving a line number prompt. For example:

```
>ENTER 10(5)↵
10          A=37.↵
>
```

CFORTRAN terminates the command, since the next computed line number would force deletion of the existing line number 15.

TRANSFERRING PROGRAMS AND STATEMENTS

CFORTRAN includes commands which permit the transfer of information from a file to CFORTRAN, from CFORTRAN to a file, and from one part of a CFORTRAN program to another. The EDIT command described here, the MERGE command, and the COPY command transfer file information to CFORTRAN; the SAVE, COPY, and MOVE commands transfer information from CFORTRAN to a file. In addition, COPY and MOVE transfer statements from one position to another within CFORTRAN.

Transferring Information From a File

To enter a program from a file, the user types the EDIT command, including a valid file name. CFORTRAN prints OK, completely clears any current program, and reads the program text from the appropriate file. Each line in the file must have a line number, and the lines must be stored in ascending line number order. The general command form is:

>EDIT file name↵

If the user enters a file name with a file name extension, CFORTRAN searches for the named file. If no file name extension is specified, CFORTRAN searches for a file with the specified name and the file name extension F4. If no such file exists, CFORTRAN then searches for a file with the specified name and no extension. If no file exists with that specified name, an error diagnostic is printed. For example,

>EDIT PROG↵

causes a search of the user's directory for the file PROG.F4. If PROG.F4 does not exist, CFORTRAN then searches for PROG. If the search is still unsuccessful, CFORTRAN prints an error message.

The MERGE command allows the user to insert corrections and additions from a file into a program currently in CFORTRAN. MERGE should not be used for initial program entry, since it is slower than EDIT. The form for the MERGE command is:

>MERGE file name↵

The file name specification and resulting searches are identical to those described above for the EDIT command.

To enter information with the MERGE command, each line in the file must have a line number. The lines may, however, be stored in any order. If any line on the file has the same line number as a line in the current program, CFORTRAN replaces the current line.

In the following example, the user lists the current program in CFORTRAN, then uses the MERGE command to merge program corrections from the file CORR.

```
>LIST ↵

1           ACCEPT X,Y
2           Z=SQRT(X**2+Y**2)
3           THETA=ATAN2(X,Y)
4           END

>
```

The file CORR contains:

```
3.8 20        FORMAT(2F10.6)
1           ACCEPT 20,X,Y
```

The user uses the MERGE command, then lists the merged program as follows:

```
>MERGE CORR ↵
OK
>LIST ↵

1             ACCEPT 20,X,Y        Line 1 from the file CORR replaces the old line 1.
2             Z=SQRT(X**2+Y**2)
3             THETA=ATAN2(X,Y)
3.8 20        FORMAT(2F10.6)       Line 3.8 read from the file is inserted
4             END                  in the proper position in the program.

>
```

In addition, the COPY command, described on page 44, permits the user to enter unnumbered program statements from a file.

Transferring Information To a File

The SAVE command allows the user to store on a file the symbolic text of his program. The relocatable binary program file is created and saved by the COMPILE or RUN command, discussed on pages 52 and 54, respectively. The COPY and MOVE commands, which are discussed below, also permit the transfer of statements from CFORTRAN to a file.

The general form of the SAVE command is

>SAVE file name↵

where the file name may include a period (.) and a three-character file name extension. If the user omits the file name extension, the last file name extension specified or implied in a SAVE or EDIT command is assigned. If no extension is specified, the implied extension is F4. The command

>SAVE↵

saves the current symbolic text on a file with the last name and extension specified or implied in a previous SAVE or EDIT command. If no file name was previously specified, CFORTRAN prints an error message.

SAVE writes the file with the line numbers right justified, leading zeros inserted, and each line number followed by a tab character. The line number field contains either five or ten characters, dependent on the length required to accommodate the longest line number in the program.

Example

```
>EDIT ABFILE↵          The user enters his program from ABFILE.F4.
OK
>MOVE 101 TO 16↵       He modifies his program.
>RENUMBER AS 10(20)↵
>SAVE↵                 The user saves the modified program.
ABFILE.F4              CFORTRAN prints the implied file name.
OLD FILE↵              The user confirms the command to write on ABFILE.F4.
OK
>QUIT↵
```

-TYPE ABFILE.F4↵ *The user lists his program in TYMEX to see exactly how it is stored.*

```
00010           TYPE 2
00030    2      FORMAT(' TYPE A, B, C,'/)
00050           ACCEPT 1,A,B,C
00070           COEFF=B**2-4.*A*C
00090           IF(COEFF)10,20,20
00110    10     TYPE 5
00130           GO TO 50
00150    20     XP=(-B+SQRT(COEFF))/(2*A)
00170           XN=(-B-SQRT(COEFF))/(2*A)
00190           TYPE 12
00210           TYPE 15,XP,XN
00230    50     CONTINUE
00250    1      FORMAT(3F9.4)
00270    5      FORMAT(' ALL ROOTS ARE IMAGINARY')
00290    12     FORMAT(2X,' X+ROOT',4X' X-ROOT')
00310    15     FORMAT(/2F10.4)
00330           END
```

The COPY and MOVE Commands

The COPY command allows the user to enter unnumbered lines from a file, store lines on a file without line numbers, list lines on the terminal; and copy lines from one part of a program to another. The MOVE command permits him to move program lines to a file or a line range.

The general form of the COPY command is:

>COPY source TO destination↵

The source may be a file name or a list of one or more line numbers or range addresses separated by commas. File name specifications are identical to those described on page 41. The destination may be a file name, T (or any other abbreviation of the word TERMINAL), or one of the following:

line number:line number

line number(increment)line number

line number(increment)

CFORTRAN does not permit both source and destination to be file names; that is,

>COPY file name TO file name↵

is not a legal command. Also, if the source is a file, the terminal is not a legal destination.

The form

>COPY source TO line number:line number↵

deletes the lines in the destination range and inserts a copy of the source lines into the destination range. Line numbers begin with the first line number specified and have as an increment the first of 1, .1, .01, and .001 which will fit the specified range. If the source is a file, no line in the file may have a line number. For example, if the file AA contains the lines

```
ACCEPT 50,X,Y
Z=ABS(X-Y)
TYPE 100,Z
```

Each line is preceded by eight blanks.

the user may copy AA to a line range in CFORTRAN as follows:

```
>COPY AA TO 50:70↵
OK
>FAST↵
50          ACCEPT 50,X,Y
51          Z=ABS(X-Y)
52          TYPE 100,Z
>
```

The form

>COPY source TO line number(increment)line number↵

is the same as the previous form, except that the line number increment is specified by the user. For example,

```
>COPY AA TO 50(10)70↵
OK
>FAST↵
50          ACCEPT 50,X,Y
60          Z=ABS(X-Y)
70          TYPE 100,Z
>
```

inserts a copy of the source lines into the program, numbered from the first line number specified with the specified increment.

When the terminating line number is not specified, the user is protected from deleting or intermixing program lines. If a copied line would be assigned a line number that would cause deletion or intermixing, a message is printed and the command is terminated. For example:

```
>COPY 3,*10,$ TO 50(10)↵
NOT ENOUGH ROOM, COMMAND NOT EXECUTED
>
```

The general form of the MOVE command is:

>MOVE line list TO destination↵

The destination can be a file name or one of the following:

line number: line number

line number(increment)line number

line number(increment)

When MOVE is executed, CFORTRAN deletes the source line list; otherwise, MOVE works exactly like COPY. For example:

```
>LIST↵

1           DO 100 I=1,N
2           D=2*C/I
3           C=C-D
4  100      CONTINUE
5           END

>MOVE 3 TO .5↵
>LIST↵

 .5           C=C-D
1             DO 100 I=1,N
2             D=2*C/I
4    100      CONTINUE
5             END

>
```

PROGRAM EDITING

When a program is in CFORTRAN, the user may modify, delete, and renumber statements, using any of the EDITOR control characters and the CFORTRAN commands described in this section.[1]

Modifying Statements

The EDIT and MODIFY commands specify a line or a group of lines to be modified. The EDIT command prints each line, then permits the user to edit it; the MODIFY command does not print the specified lines prior to allowing the modification.

The EDIT command has the general form

>EDIT line address list↵

where the addresses may be any of the forms described on page 36.

If only one line is addressed in the EDIT command, the line is printed on the terminal, providing a line image for editing purposes.

Example

```
>EDIT 2↵
2           S=A-B/2
2           S=(A+B+C)/2↵
>
```

If more than one line is addressed in the EDIT command, CFORTRAN prints the lines addressed one at a time, waits for the user to edit that line, and then continues to the next line. When the last line is edited, control returns to CFORTRAN command level.

1 - For a complete description of control characters, see the Tymshare EDITOR Reference Manual.

Example

```
>EDIT 1,5:$↵
1          ACCEPT 300,A,B
1 H^C      ACCEPT 300,A,B,C↵
5 200      FORMAT(3F10.2)
5 Z^C(     FORMAT(4D^CF10.2)
6 300      FORMAT(3F10.4)
6 Z^C1 300    FORMAT(3F12D^C.4)
7          STOP
7 I^C      END↵
>
```

The MODIFY command has the form:

>MODIFY line address list↵

The MODIFY command is identical to the EDIT command, except that the lines addressed are not printed on the terminal before allowing the modification.

Deleting Statements

The user may delete any part of his program with the DELETE command or all of his program with the CLEAR command.

The DELETE command deletes any line or group of lines, and has the general form

>DELETE line address list↵

where the list may contain one or more line or range addresses separated by commas.

Examples

>DELETE *12↵

deletes the 12th line;

>DELETE .,200:200+10,*400↵

deletes the current line, line number 200 through the 10th line following line number 200, and the 400th line in the program.

In the second example above, the address *400 refers to the 400th line before execution of the DELETE command, not the 400th line after the lines addressed by . and 200:100+10 are deleted.

The CLEAR command simply erases the entire program. It has the general form:

>CLEAR↵

CFORTRAN replies:

ALL?

This question may be answered by typing Y (for Yes) or N (for No), followed by a Carriage Return. If Y is typed, CFORTRAN erases the program; if N is typed, the command is aborted. For example,

```
>CLEAR↵
ALL? Y↵
```

erases the entire program.

Renumbering Statements

The user may change all or part of the line numbers in his program with the RENUMBER command. The general form is

>RENUMBER [source line range] [AS new line range]↵

where everything within brackets is optional. If the user types only

>RENUMBER↵

CFORTRAN renumbers the entire program, starting with line number 1 and selecting the largest increment from 1, .1, .01, and .001 which permits all lines to have legal line numbers. For example:

```
>LIST↵

 2            DIMENSION X(10),Y(10),Z(10),W(10)
 2.7          DO 10 I=1,10
 2.9          Z(I)=Z(I)+Y(I)
 3.3          W(I)=(X(I)+Y(I))**2
 5   10       CONTINUE
11            END

>RENUMBER↵
>LIST↵

1          DIMENSION X(10),Y(10),Z(10),W(10)
2          DO 10 I=1,10
3          Z(I)=Z(I)+Y(I)
4          W(I)=(X(I)+Y(I))**2
5 10       CONTINUE
6          END

>
```

When the user wishes to renumber part of the program and permit CFORTRAN to select the line number increment, he types:

>RENUMBER source line range↵

The new line numbers start with the first line number in the source line range specification. If possible, a line number increment is selected so that no line in the program is deleted; otherwise, CFORTRAN prints an error diagnostic. For example, using the original program on the preceding page:

```
>RENUMBER 2:3.3↵
>LIST↵

 2          DIMENSION X(10),Y(10),Z(10),W(10)
 2.1        DO 10 I=1,10
 2.2        Z(I)=Z(I)+Y(I)
 2.3        W(I)=(X(I)+Y(I))**2
 5   10     CONTINUE
11          END

>
```

When the user wants to renumber the entire program with specified line numbers, he uses the form:

>RENUMBER AS new line range↵

The new line range specification may be any of the following:

line number:line number

line number(increment)line number

line number(increment)

For example:

```
>RENUMBER AS 20:30↵
>LIST↵

20          DIMENSION X(10),Y(10),Z(10),W(10)
21          DO 10 I=1,10
22          Z(I)=Z(I)+Y(I)
23          W(I)=(X(I)+Y(I))**2
24 10       CONTINUE
25          END

>
```

The same renumbering occurs if the user types:

```
>RENUMBER AS 20(1)30↵
```

or

```
>RENUMBER AS 20(1)↵
```

RENUMBER does not delete or rearrange lines. If the specification requires deletion or rearrangement of lines, the command is not executed, and CFORTRAN prints an error diagnostic.

Examples

```
>RENUMBER 20:24 AS 20(2)↵
CANNOT RENUMBER, CHECK LINE RANGE
>
```

```
>RENUMBER 20:24 AS 20(.5)↵
>FAST↵
20          DIMENSION X(10),Y(10),Z(10),W(10)
20.5          DO 10 I=1,10
21          Z(I)=Z(I)+Y(I)
21.5          W(I)=(X(I)+Y(I))**2
22 10       CONTINUE
25          END
>
```

COMPILATION AND EXECUTION

The user may compile, execute, and debug his program within CFORTRAN, and may include any of the switches available in TYMEX.[1] A file does not have to be read into CFORTRAN before compilation or execution; the COMPILE and RUN commands automatically access the files. For editing purposes, however, the file must be accessed with the EDIT command discussed on page 41.

The CFORTRAN COMPILE and RUN commands are identical to the TYMEX COMPILE and FDEBUG commands, respectively.

1 - See the Tymshare TYMEX Reference Manual.

The COMPILE Command

The COMPILE command compiles a program, prints any error diagnostics, and writes a relocatable binary file. No compilation occurs when a relocatable file newer than the source file already exists. The general form is:

>COMPILE [file name,file name,...]↵

The COMPILE command accesses each file and creates a relocatable binary file for each. Each relocatable file has the name of the symbolic file with the file name extension .REL. For example,

>COMPILE AFILE,BFILE,CFILE↵

accesses and compiles the files, then saves the corresponding binary programs on the files AFILE.REL, BFILE.REL, and CFILE.REL.

The most recent file name argument in a COMPILE or RUN command is retained for subsequent COMPILE or RUN commands.[1]

>COMPILE↵

assumes the argument of the most recent COMPILE or RUN command, or if no prior COMPILE or RUN command has established a file name argument, compiles the current text. An interim CLEAR or EDIT command erases the file name argument.

The current text is indicated by a dollar sign ($) entered as a file name. For example,

>COMPILE $↵

compiles only the current text. The command

>COMPILE ABC,DEQ,$,FILEX↵

compiles the current text and the files ABC, DEQ, and FILEX.

Example

```
>COMPILE QUADR,AFILE↵
F-IV:  QUADR.F4
 MAIN.
F-IV:  AFILE.F4
 ASUB
```

1 - The RUN command is described on page 54.

```
>EDIT QUADR↵                    The EDIT command erases the file name
OK                              specification of the COMPILE command.
>EDIT 94↵
94 15       FORMAT(/2F10.4)
94 15       FORMAT(/2F8.3)↵
>COMPILE↵                       This COMPILE command instructs CFORTRAN
F-IV:  008XXS.TMP               to compile the current text.
 MAIN.

>
```

If the symbolics being compiled are currently saved on a file, the relocatable binary program is saved on a file with the same name and the file name extension .REL.

Example

```
>10         ACCEPT 10,A,B,C↵
>15 10      FORMAT(3F8.3)↵
>20         D=SQRT(A**B/2*C)↵
>30         ANS=EXP(D)↵
>35         TYPE 20,ANS↵
>40 20      FORMAT(F12.4)↵
>45         END↵
>SAVE EXPD↵                     The symbolics are saved on the file EXPD.F4.
NEW FILE↵
OK
>COMPILE↵                       The compiled program is saved on the file EXPD.REL.
F-IV:  EXPD.F4
 MAIN.

>
```

If the symbolics being compiled are not currently saved on a file, the symbolic program and the relocatable program are assigned temporary storage. A subsequent SAVE command saves the current symbolics and, if it exists, the corresponding relocatable binary program.

Example

```
-CFORTRAN↵

>EDIT DEXP↵
OK
>LIST↵

10          ACCEPT 10,A,B,C
15 10       FORMAT(3F8.3)
20          D=SQRT(A**B/2*C)
30          ANS=EXP(D)
35          TYPE 20,ANS
40 20       FORMAT(F12.4)
45          END

>MOVE 15 TO 37↵
>RENUMBER AS 10(5)↵
>COMPILE↵
F-IV:  008XXS.TMP
 MAIN.

>SAVE DEXP↵
OLD FILE↵
OK
>
```

The symbolics are saved on the file DEXP.F4, and the corresponding relocatable binary program is saved on the file DEXP.REL.

The RUN Command

The RUN command compiles and loads the specified program and transfers control to DEBUG. The user then controls execution using DEBUG.[1] He may return to CFORTRAN command level by typing EDIT and a Carriage Return in DEBUG. If the user's program successfully runs to completion, control automatically transfers from DEBUG to the CFORTRAN command level.

The general form of the RUN command is:

>RUN [file name,file name,...]↵

RUN compiles the files, if necessary, and creates one relocatable file for each symbolic file named. It then loads the files as one program and starts DEBUG. The compile function of the RUN command is identical to the COMPILE command function; file name arguments are established in the same manner. RUN, like COMPILE, does not access a file for editing purposes.

1 - See the Tymshare DEBUG Reference Manual.

Example

```
-CFORTRAN↵

>RUN ABC,AFILE,SUB1↵
F-IV:   ABC.F4          A relocatable binary file is created for each source file;
 MAIN.                  that is, ABC.REL, AFILE.REL, and SUB1.REL are
F-IV:   AFILE.F4        created. The files specified constitute one program.
 ASUB
F-IV:   SUB1.F4
 BSUB
LOADING

DEBUG!
*
```

To initiate execution in DEBUG, the user types START followed by a Carriage Return.

Setting Switches

CFORTRAN permits the user to set any of the switches available with the TYMEX EXECUTE command.[1] Switches may be preset to apply to every file named in all COMPILE and RUN commands. In addition, switches may be included in the command to apply to one, part, or all of the files in the file name list.

To preset switches, the user types:

>SWITCH /switch name /switch name ...↵

The switches apply to every file in all COMPILE and RUN commands unless a contradictory switch is set, a new SWITCH command is entered, or all switches are nullified. The command

>SWITCH↵

cancels all switches previously set with a SWITCH command.

Within a COMPILE or RUN command, switches may be specified in three ways: preceding the entire file name list, immediately following a file name, or embedded in the file name list. The user may use any combination of the specification. If a /switch precedes the entire file name list, that is,

>RUN /switch file name,file name,file name,...↵

1 - See the Tymshare TYMEX Reference Manual.

the switch is set for each file named. A switch immediately following a file name applies only to that file. For example,

```
>COMPILE ABC,DEQ/CREF,FILEX↵
```

sets the CREF switch for file DEQ only. A switch specification which appears within the file name list and is set off with commas or spaces applies to the files whose names follow the switch name. For example,

```
>COMPILE AFILE,ABC,/LIBRARY,DEQ,FILEX↵
```

sets the LIBRARY switch for files DEQ and FILEX.

NOTE: When two contradictory switches are set, the last switch specified is used.

The file name argument assumed by a subsequent COMPILE or RUN command does not include the switches which precede the file name list; a switch specified in either of the other two ways is included. For example:

```
>COMPILE /N QUADR,AFILE/LIBRARY,/I,FILEX,B↵
>RUN↵
```

The file name argument used for the RUN command includes the /LIBRARY switch and the /I, but not the preceding /N switch. The user may, therefore, include preceding switches in the RUN command without changing the implied file name argument. For example, if the user types

```
>COMPILE /N QUADR,AFILE/LIBRARY,/I,FILEX,B↵
>RUN /LIST↵
```

CFORTRAN interprets the RUN command as:

```
>RUN /LIST QUADR,AFILE/LIBRARY,/I,FILEX,B↵
```

SECTION 5

FORTRAN IV LIBRARY

The FORTRAN IV library described in the Tymshare TYMCOM-X FORTRAN IV Reference Manual has been modified and expanded. The new functions and subroutines used for a specific purpose, such as file handling, are included in the description of the appropriate capability. All other new or modified subprograms are described in this section.

In addition to the FORTRAN IV library, the user may access any routine in the Scientific Subroutine Package.[1] When naming his program file(s) in a COMPILE, LOAD, EXECUTE, DEBUG, or TRY command, the user includes (UPL)SSP/LIB in the file name list. For example:

```
-EXECUTE PROG,(UPL)SSP/LIB
```

The FORTRAN IV library subroutines discussed in this section are listed in the following table.

Name	Description
AFILE	Opens file for input across user boundaries.
BENCHM	Prints CPU and elapsed time.
BFILE	Opens file for input or output across user boundaries.
DCLOSE	Closes a previously opened direct access file.
DDEL	Deletes a specified record from a direct access file.
DEFINB	Opens a fixed length record direct access file.
DGROW	Increases the number of records on a direct access file.
DOPENB	Opens a direct access file.
DREAD	Reads the requested information from a direct access file.
DSKERR	Prints the interpretation of a direct access file error code.
DWRITE	Writes the requested information to a direct access file.
DYNARY	Dynamically allocates array storage.
EXIT	Completes output and terminates execution.
IFILE	Opens a sequential file for input.
NAMPPN	Converts a user name to a file directory number.
OFILE	Opens a sequential file for output.

1 - For a complete list of the routines in the Scientific Subroutine Package, see the Appendix.

Name	Description
ONECHR	Prints a right-justified ASCII character on the terminal.
PLOT	Plots values on CalComp plotter.
PLOTF	Initializes program for off-line plotting with CalComp plotter.
PLOTS	Initializes common variables and initiates a CalComp plotter.
RENAME	Renames or deletes specified files.
RUN	Transfers control to another core image program.
SCALE	Determines scaling factor for CalComp plotting.
SYMBOL	Draws one character or a string of characters on a CalComp plotter.
TIMER	Specifies maximum CPU time for program execution.
WHERE	Returns the pen location and factor for the CalComp plotter.

ARGUMENTS

The subroutine calling arguments are described with the respective subroutines; however, many of the subroutines have common arguments. These arguments are described below.

file number — An integer constant or variable corresponding to the number of the file being accessed. Valid file numbers are 1 through 29. A maximum of 15 files may be open simultaneously.

'file name' — A string constant or variable representing the name of the file being opened. This argument may be the actual file name enclosed in single quote marks, a double precision variable name, or an array name. The file name is one to ten characters in length and may include as many as six characters plus a period (.) and a three-character file name extension. If the file name is assigned to an array and fewer than ten characters are specified, the first array element with fewer than five characters terminates the file name. For a file name exactly five characters in length, the next array element must be set to zero or blank. For example, if the array NAME has been dimensioned for at least two elements, each of the following specifies a file name.

```
NAME(1)='FILEA'
NAME(2)='B'
```

```
NAME(1)='FILEA'
NAME(2)=' '

NAME(1)='FILEA'
NAME(2)=0

NAME(1)='FILEA'
NAME(2)='B.EXT'
```

Regardless of the way the file name is specified, the file name read includes only the characters preceding the first blank. For example,

```
DOUBLE PRECISION NAME
NAME='FILE A.NEW'
```

is read as:

```
NAME='FILE'
```

user name — A variable name or constant indicating the user name associated with the file being opened. If the argument has the integer value 0, the program assumes the current user name. If the user name is an array variable, it may contain as many as 12 characters, and the first unused array element must be set equal to zero, or three array elements must be specified. For example, each of the following sets of statements specifies a user name.

```
DIMENSION UNAME(3)
UNAME(1)='JOHND'
UNAME(2)='OECOR'
UNAME(3)='P'

DIMENSION UNAME(3)
UNAME(1)='JOHND'
UNAME(2)='OE'
UNAME(3)=0

DIMENSION UNAME(3)
UNAME(1)='JOHND'
UNAME(2)='OE'
UNAME(3)=' '
```

NOTE: This user name description is valid only for those subroutines which permit a double precision user name.

MODIFIED SUBROUTINES

Some of the subroutines discussed in the Tymshare TYMCOM-X FORTRAN IV Reference Manual have been changed. In the DEFINE, FILE, IFILE, and OFILE subroutines, the file name argument has been changed to accommodate file names of the form described on page 58. The EXIT subroutine now completes any unfinished output procedure and releases all files before terminating program execution. The function of the CHAIN subroutine has been replaced with the program segmentation capability described on page 17.

MULTIPLE ENTRY POINT PROGRAM TRANSFERS

FORTRAN programs may call the RUN subroutine to access other FORTRAN programs or Tymshare Library programs. In addition, a called FORTRAN program may contain a statement which defines multiple entry points; the desired entry point is specified as an argument in the CALL RUN statement.

The RUN subroutine is called by a statement of the form

CALL RUN('device name','file name',user name,entry number)

where the arguments are as defined below.

'device name'	The name of the device on which the program is stored; this argument may be 'DSK', 'SYS', or a variable containing the device name. 'DSK' accesses any user program; 'SYS' accesses a Tymshare Library program.
'file name'	The name of the program file. This argument may be the actual file name enclosed in single quote marks or the name of the variable containing the file name. The file named must be a core image file; the system assumes the extension .SAV or .SHR.
user name (optional)	The user name as described on page 59. If the CALL RUN statement accesses a Tymshare Library program, the user name must be equal to zero.
entry number (optional)	An integer constant or variable defining the entry point in the program being called. An entry number equal to n corresponds to the n^{th} statement label in the /ENTRY statement described below. If this argument is omitted or is equal to zero, execution of the called program begins with the first statement in the program.

NOTE: If an entry number is specified, a user name, which may be zero, must also be specified.

If the user wishes to begin execution of a called program at selected points within the program, he may include a /ENTRY statement as the first statement in the main program of the called program. The form of this statement is

/ENTRY a,b,c,...

where a,b,c,... are statement labels corresponding to statement labels in the called main program. Statement a is entry point 1, statement b is entry point 2, etc. For example, assume that the calling program contains the statement

CALL RUN('DSK','PROG2',0,3)

and that the called program, PROG2, contains the statement

/ENTRY 7,15,24,9,6

as the first statement of the main program. When FORTRAN encounters the CALL RUN statement, core is cleared, PROG2 is loaded, and the program execution begins with the statement labeled 24 in PROG2, which is entry point 3.

NOTE: The entry point argument in the CALL RUN statement must be less than or equal to the number of statement labels in the /ENTRY statement. The user's program should check for this possible problem, as it is not automatically checked by the system.

Example

```
-TYPE SQUARE↵
```

```
        TYPE 10
10      FORMAT(' THIS IS THE CALLING PROGRAM')
        TYPE 11
11      FORMAT(' INPUT I1, I2',/)
        ACCEPT 13,I1,I2
13      FORMAT(2I3)
        DO 20 I=I1,I2
        J=I**2
20      TYPE 30,I,J
30      FORMAT(2I5)
        IENT=1
        IF(J.LE.400)IENT=2
        IF(J.GT.800)IENT=0
        IF(J.LE.250)IENT=3
        CALL RUN('DSK','SQUAD',0,IENT)
        END
```

The entry point selected depends upon the computed value of J.

This statement transfers control to program SQUAD in the current user's directory and begins execution at entry point IENT.

```
-TYPE SQUAD↵
```

If the entry point is 0, execution starts at the beginning of the program; if the entry point is 1, execution begins at statement 30; if the entry point is 2, execution begins at statement 40; and if the entry point is 3, execution begins at statement 60.

```
        /ENTRY 30,40,60
        REAL LENGTH
        TYPE 7
7       FORMAT(' THIS IS THE CALLED PROGRAM, SQUAD')
30      TYPE 2
2       FORMAT(13H TYPE A, B, C,/)
        ACCEPT 1,A,B,C
1       FORMAT(3F9.4)
        IF(COEFF)10,20,20
10      TYPE 5
5       FORMAT(' ALL ROOTS ARE IMAGINARY')
        GO TO 50
20      XPOSRT=(-B+SQRT(COEFF))/(2.*A)
        XNEGRT=(-B-SQRT(COEFF))/(2.*A)
        TYPE 12
12      FORMAT(2X,7H X+ROOT,4X,7H X-ROOT)
        TYPE 15,XPOSRT,XNEGRT
15      FORMAT(/2F10.4)
40      TYPE 3
        TYPE 16
3       FORMAT(' THIS IS ENTRY POINT TWO IN SQUAD')
16      FORMAT(' INPUT X1,Y1,X2,Y2',/)
        ACCEPT 4,X1,Y1,X2,Y2
4       FORMAT(4F9.4)
        LENGTH=SQRT((X1-X2)**2+((Y1-Y2)**2))
        SLOPE=(Y1-Y2)/(X1-X2)
        TYPE 25,LENGTH,SLOPE
25      FORMAT(' LENGTH= ',F8.3,'    SLOPE= ',F8.3)
60      TYPE 61
61      FORMAT(' THIS IS ENTRY POINT THREE IN SQUAD')
        TYPE 67
67      FORMAT(' INPUT TIME, P,Q',/)
        ACCEPT 62,TIME,P,Q
62      FORMAT(3F8.2)
        DO 65 N=1,5
        R=(P*Q+N)/TIME
65      TYPE 66,N,R
66      FORMAT(I2,F9.3)
50      CONTINUE
        END
```

```
-LOAD SQUAD↵
LOADING
5K CORE
```

The user loads the program to create a core image file.

```
-SAVE SQUAD↵
JOB SAVED
```

He saves the core image file. The system names file file SQUAD.SAV.

```
-EXECUTE SQUARE↵
LOADING
EXECUTION

THIS IS THE CALLING PROGRAM
INPUT I1, I2
 14 19↵

  14  196
  15  225
  16  256
  17  289
  18  324
  19  361
THIS IS ENTRY POINT TWO IN SQUAD
INPUT X1,Y1,X2,Y2
14.5 23.4 19.2 44.1↵

LENGTH=   21.227     SLOPE=    4.404
THIS IS ENTRY POINT THREE IN SQUAD
INPUT TIME, P,Q
22.3 247.8 432.2↵

1 4802.698
2 4802.743
3 4802.787
4 4802.832
5 4802.877
EXIT

-
```

THE BENCHM SUBROUTINE

The BENCHM subroutine causes FORTRAN to print the CPU time and elapsed time for the program execution to that point. BENCHM is called without arguments; for example:

CALL BENCHM

THE AFILE SUBROUTINE

The AFILE subroutine allows the reading of files from other users' directories. The form of the CALL statement is

CALL AFILE(file number,'file name','user name')

where the argument specifications are as described on page 58.

THE BFILE SUBROUTINE

The BFILE subroutine allows files to be accessed from other users' directories for input or output. The statement is

CALL BFILE(file number,'file name','user name',access mode)

where the access mode is the word INPUT or OUTPUT in single quote marks, or a variable name containing the word INPUT or OUTPUT. The other arguments are described on page 58.

THE DYNARY SUBROUTINE

DYNARY is a dynamic array storage allocation subroutine. The user may vary the array sizes to accommodate the desired compilations. For example, a program which normally computes and has allocated for 25 array values may be called by DYNARY and the array expanded to 2000 elements. The call to DYNARY is

CALL DYNARY(IERR,SUBR,s_1,s_2,...,s_n,0,par_1,par_2,...,par_m)

where the arguments are defined as follows.

IERR — An integer variable returned as:

1. The number of words of core available if used as the only argument or if there is insufficient core to allocate arrays of the specified size.
2. -1 if a RETURN is encountered in the called subprogram; a RETURN releases the dynamically allocated arrays. The called subprogram need not contain a RETURN statement.

SUBR — The subprogram which uses the arrays to be allocated. The CALL DYNARY executes the equivalent of the call to the subprogram; that is,

CALL SUBR($array_1$,$array_2$,...,$array_n$,par_1,par_2,...,par_m)

where $array_1$,$array_2$,...,$array_n$ are the dynamically allocated arrays, and par_1,par_2,...,par_m are the other arguments for SUBR. SUBR must be declared in an EXTERNAL statement.

$s_1, s_2, \ldots, s_n, 0$ — The variable names whose values are the sizes, in words, of the arrays $array_1, array_2, \ldots, array_n$ used by the subprogram SUBR. The zero (0) argument terminates the list of array sizes.

$par_1, par_2, \ldots, par_m$ — The arguments for SUBR other than the dynamically allocated arrays. These arguments are passed, unchanged, to SUBR.

NOTE: All dynamically allocated arrays are deleted when any segment of the program is replaced by an overlay.

If a file is opened after the dynamic allocation of arrays, it must be closed before a RETURN, as the input/output buffers are deleted with the dynamically allocated arrays.

Dynamically allocated arrays cannot be declared in a COMMON statement.

THE NAMPPN SUBROUTINE

The NAMPPN subroutine converts a user name to a file directory number. This number has the form of two integers separated by a zero. The NAMPPN subroutine is called with the statement

CALL NAMPPN('user name',variable name,error)

where the user name specification is as described on page 59, the variable name is the variable to which the file directory number is assigned, and the error codes are as listed in the table below.

Error Code	Meaning
0	Successful operation
1	No channels available
2	Bad user name argument
3	Not a valid user name on this system
11	System error

THE ONECHR SUBROUTINE

The user may instruct FORTRAN to print on the terminal the ASCII character equivalent of the right-most eight bits of a variable value with no system conversion.[1] Thus, the user may access any ASCII character. The user calls this subroutine with the statement

```
CALL ONECHR(x)
```

where x is a variable name. For example,

```
I=35
CALL ONECHR(I)
```

causes the system to print an equal sign (=), because 35 is the external decimal equivalent of the ASCII character =.

THE RENAME SUBROUTINE

The RENAME subroutine permits the user to change the name and/or protection code on an existing file and to delete files. The form is

```
CALL RENAME('on','oe','nn','ne',np,error)
```

where
- on is the old file name.
- oe is the old extension.
- nn is the new file name.
- ne is the new extension.
- np is the protection code of the new file.[2]
- error is an error code returned by the subroutine.

NOTE: The argument for the error code must be a variable name, not a constant.

To delete a file, the arguments nn, ne, and np are specified as zero. For example,

```
CALL RENAME('TEST','IMP',0,0,0,IERR)
```

deletes the file named TEST.IMP.

1 - See page 133 of the Tymshare TYMCOM-X FORTRAN IV Reference Manual for a table of ASCII characters.

2 - For a description of file protection codes, see the Tymshare TYMEX Reference Manual.

The error codes are listed below. If a nonzero error code is returned, it indicates that no renaming or protection changes were made.

Code	Description
0	Successful completion of subroutine
1	User file directory not found
2	Protection failure
3	File in use by another job
4	New file name already in use
5	No file selected on rename channel
6	Old file not found
7	Disk error

A maximum of five characters is allowed in the file names; a maximum of three nonblank characters is allowed in the extensions. If an extension is blank, a zero argument or a string of at least three blanks should be used. If no protection change is required, a zero argument should be used. The protection code must be specified as an octal number, such as 057, 055, or 077.

THE TIMER SUBROUTINE

The TIMER subroutine permits the user to specify the maximum number of CPU seconds his program can run. If the program runs longer than the specified time, a message is printed, execution is terminated, and control is returned to TYMEX command level.

TIMER is called by the statement

CALL TIMER(n)

where n is an integer equal to the maximum CPU time in seconds.

DIRECT ACCESS FILE SUBROUTINES

The TYMCOM-X FORTRAN IV library contains seven direct access file (DAF) subroutines: DOPENB, DREAD, DWRITE, DDEL, DCLOSE, DGROW, and DSKERR. The DAF subroutines have been superseded by the Tymshare file handling capability described on page 3, but are retained in the FORTRAN library for compatibility. DAF processing is valid only on

files that have been created by the DAF subroutines; however, it is possible to read data from a sequential file and to store it on a direct access file. Binary files created with the DAF subroutines are compatible with the Tymshare file handling capability.

Arguments

With the exception of DOPENB, which requires additional arguments, the user calls each DAF subroutine with one or more of the parameters below as arguments. Note that missing arguments or arguments out of sequence cause an error.

N The file number, as described on page 58.

E An integer variable name whose value is set by the DAF subroutines to indicate success or failure of operation. It is important for the user to check error codes after DAF calls to ensure proper operation of the program. The variable E is assigned a value of 1 if the operation is successful; any other value indicates a failure. See page 71 for a complete list of codes and their meanings.

R The associated variable which has an integer value indicating the number of the record to be accessed. Note that this argument must be an integer variable. DOPENB automatically assigns a value of 1 to this variable after a successful operation in DOPENB; and after each subsequent operation by DWRITE, DREAD, or DDEL, the subroutine automatically increments this parameter by 1.

A An array name specifying the array to be used in a data transfer. The array dimensions must be large enough to accommodate at least one record.

The DOPENB Subroutine

The user must open each file with the DOPENB subroutine before any direct access file activity can occur. DOPENB may be used to create a file or open an existing file. The user calls DOPENB with the statement

```
CALL DOPENB(N,E,R,F,USER,MODE,RMAX,L)
```

where N, E, and R are defined above, and the other arguments are defined as follows.

F The file name argument as described on page 58.

USER The user name argument as described on page 59.

MODE An integer from 0 to 3, inclusive, indicating the activity intended. The different values indicate the following:

0 File initialization. After the file has been initialized, MODE is set to 2 internally. Mode 0 is valid only if no file with the specified name, F, exists.

1 Read only; file used for input.

2 Write only; file used for output.

3 Read, write, and update; file used for input, output, or updating.

To change MODE, the user must close the file with the DCLOSE subroutine and use DOPENB to reopen it with the new MODE desired.

RMAX An integer variable name or constant whose value represents the maximum number of records on the file. This value cannot be changed in subsequent DOPENB statements. Records not written actually exist as undefined records. A constant argument is used only for file initialization. Any reference to an existing file must have a variable name argument; DOPENB assigns the corresponding value for the specified file.

L An integer variable name or constant whose value represents the actual record length in five-character storage words. All records must be of fixed length. A constant argument is used only for file initialization. Any reference to an existing file must have a variable name argument; DOPENB assigns the corresponding value for the specified file.

The following example statement opens a new file:

```
CALL DOPENB(1,IE,R,'FILE.NEW',0,0,NREC,LNTH)
```

When the program encounters this statement, the file, FILE.NEW, is opened as file number 1 and initialized under the current user name. The file contains NREC records of length LNTH. The mode is changed to 2, R is automatically set to 1 to indicate the first record, and IE is set to 1 to indicate a successful operation. Within this FORTRAN program, all activity on file number 1 refers to this file until file number 1 is closed with the DCLOSE subroutine.

DOPENB may also be used to open an existing file in the following manner:

```
DOUBLE PRECISION NAME
NAME = 'AFILE'
CALL DOPEN(6,IE,IR,NAME,ABC123,1,10,4)
```

These instructions cause a search of user ABC123's file directory for the existing file, AFILE. If the file is found and a read operation is permitted, IE is set to 1 to indicate success, and IR is initialized to 1. The file AFILE is open as file number 6 and contains ten records, four words long. If the file is not found or its use is prohibited, IE is set to indicate an error, and control is returned to the user's program.

The DREAD Subroutine

The DREAD subroutine reads the requested record into the array indicated. The user calls DREAD with the statement:

```
CALL DREAD(N,E,R,A)
```

DREAD locates record R on file number N and copies the data for that record into the array A. The variable E is automatically set to the appropriate value. If the read is successful, E is set to 1, and the record number is incremented by 1. The MODE of file number N must be set in the DOPENB subroutine to permit a read operation.

The DWRITE Subroutine

The DWRITE subroutine transfers data from the named array to the specified record of the file number indicated. The CALL statement has the form:

```
CALL DWRITE(N,E,R,A)
```

The data in array A is written on record R of file number N, E is set to the proper value, and R is incremented by 1.

The DDEL Subroutine

The DDEL subroutine permits the user to delete a specified record. The record is marked as undefined and is inaccessible to all DAF subroutines. The form of the CALL statement is:

```
CALL DDEL(N,E,R,A)
```

DDEL locates record R in file number N, reads the contents of that record into array A, and marks the record as undefined. For example:

```
DIMENSION ALPHA(4)
IR=4
CALL DDEL(7,IE,IR,ALPHA)
```

Record 4 on file number 7 is marked as undefined; ALPHA contains the previous contents of record 4.

NOTE: The user may write over the contents of a record without deleting the record.

The DCLOSE Subroutine

The function of the DCLOSE routine is to close a particular file. All open files must be closed before program termination. Files open when EXIT is called remain open, resulting in attempts being made to open files that are already open when the program is rerun. Except for DOPENB, no DAF subroutine can be used with a closed file. The CALL statement to access DCLOSE is:

CALL DCLOSE(N,E)

The file currently open as file number N is closed and a new file number N may be opened, or the previously open file may be reopened as any unused file number.

The DGROW Subroutine

The DGROW subroutine enables the user to change the size of an existing DAF file by changing the maximum number of records. The CALL statement is

CALL DGROW(N,E,RMAX,A)

where N, E, and A are as described on page 68, and RMAX is the maximum number of records in the file.

The DSKERR Subroutine

The DSKERR subroutine interprets error indicators. The CALL statement is:

CALL DSKERR(E)

The error message corresponding to the value of E is printed on the terminal. The error messages are listed in the table below.

Error Code	Printed Message
-3, -2	Impossible disk handler error (system error; call your Tymshare representative)
1	(No message; successful operation)

Error Code	Printed Message
2	Invalid data set reference number
3	Insufficient core for program
4	Mode incompatible with existing files
5	File not accessible
6	Requested file does not exist
7	Device output error
8	Device input error
9	Input/output requested on closed data set
10	End of file on requested data set
11	Unsuccessful open attempt
12	Attempt to initialize active file
13	Unsuccessful close attempt
14	Associated variable invalid
15	Attempt to read an undefined record
16	No available input/output channel
17	Attempt to open data set already open
18	Attempt to write a read-only data set
19	Attempt to read a write-only data set
20	Invalid logical record length
21	Invalid attempt to delete logical record

The following example is a program using six DAF subroutines.

```
-TYPE DAFEX↵

      INTEGER E,R,RMAX,N(2)          N is the transfer array.
      DOUBLE PRECISION NAME
      TYPE 10
10    FORMAT(' ENTER FILE NAME! ',$)
      ACCEPT 12,NAME
12    FORMAT(A10)
      CALL DOPENB(1,E,R,NAME,0,0,5,2)   The user creates a file on data set 1.
      IF(E.EQ.1)GO TO 11                The file will have a maximum of
      IF(E.EQ.12)GO TO 50               five records, each two words long.
      TYPE 101,E
```

```
101     FORMAT(' ERROR ',I2,' IN FILE INITIALIZATION')
11      DO 1 I=5,25,5
        N(1)=I
        N(2)=-I
        CALL DWRITE(1,E,R,N)
1       CALL DSKERR(E)
        CALL DCLOSE(1,E)
        CALL DSKERR(E)
50      CALL DOPENE(1,E,R,NAME,0,3,RMAX,L)
        CALL DSKERR(E)
        TYPE 2,RMAX,L
2       FORMAT(' # OF RECORDS=',I2,/' #OF WORDS PER RECORD=',I2)
        DO 20 I=1,RMAX
        CALL DREAD(1,E,R,N)
        IF(E.NE.1)GO TO 60
        IB=R-1
        TYPE 4,IB,(N(J),J=1,L)
4       FORMAT(' RECORD #',I1,'   ',4I4)
        GO TO 20
60      CALL DSKERR(E)
        IB=R-1
        TYPE 102 , IB
102     FORMAT(' RECORD #',I2//)
        IF(E.EQ.15)GO TO 20
        CALL DCLOSE(1,E)
        CALL DSKERR(E)
        CALL EXIT
20      CONTINUE
        R=2
        CALL DDEL(1,E,R,N)
        IF(E.EQ.1)GO TO 70
        CALL DSKERR(E)
70      IB=R-1
        TYPE 104,IB
104     FORMAT(' RECORD #',I2,' DELETED')
        TYPE 103
103     FORMAT(//' ASK FOR SELECTED RECORDS;ENTER ZERO TO STOP.')
40      TYPE 6
6       FORMAT(//' WHICH RECORD?? '$)
        ACCEPT 7,R
7       FORMAT(I1)
        IF(R.EQ.0)GO TO 30
        CALL DREAD(1,E,R,N)
        CALL DSKERR(E)
        IF(E.NE.1)GO TO 40
        IB=R-1
        TYPE 4,IB,(N(J),J=1,L)
        GO TO 40
30      CALL DCLOSE(1,E)
        CALL DSKERR(E)
        CALL EXIT
        END
```

The user closes the file on data set 1 so that he may reopen it with MODE set to read, write, or delete.

R is automatically incremented by 1; to print the number of the record just read, the user must subtract 1.

The user should close each DAF file before calling EXIT.

```
-EXECUTE DAFEX↵
F-IV:  DAFEX
 MAIN.
LOADING
EXECUTION

ENTER FILE NAME! DAFILE↵

# OF RECORDS= 5
#OF WORDS PER RECORD= 2
RECORD #1      5  -5
RECORD #2     10 -10
RECORD #3     15 -15
RECORD #4     20 -20
RECORD #5     25 -25
RECORD # 2 DELETED


ASK FOR SELECTED RECORDS;ENTER ZERO TO STOP.


WHICH RECORD? 4↵

RECORD #4     20 -20


WHICH RECORD? 2↵

ATTEMPTING TO READ AN UNDEFINED RECORD


WHICH RECORD? 3↵

RECORD #3     15 -15


WHICH RECORD? 0↵

EXIT
```

CALCOMP PLOTTER SUBROUTINES

The CalComp plotting package consists of eight subroutines. The PLOTS subroutine initializes the plotter and must be the first plotting subroutine called.

The sign conventions for the CalComp plotter subroutines are designed for standard report-type graphs, with the vertical dimension designated as the Y axis and the horizontal dimension as the X axis. Standard graphs are usually presented on 8-1/2" by 11" or 11" by 17" sheets. The positive Y

axis is directed to the left across the plotter drum, parallel to the pen carriage. The positive X axis is directed back into the plotter toward the paper supply roll.

The AXIS Subroutine

The AXIS subroutine sets up one axis, including scales and labels. The calling sequence is

```
CALL AXIS(X,Y,LABEL,LCNT,LENGTH,ANGLE,X0,DX)
```

where X and Y are the coordinates of the axis origin.

LABEL is the array containing the axis label.

LCNT is a constant or variable equal to the number of characters in the label.

LENGTH is the axis length in inches.

ANGLE is the orientation of the axis in degrees measured counter-clockwise from the horizontal

X0 is the axis value at the origin.

DX is the difference in value between two adjacent points on the axis.

The LINE Subroutine

LINE draws a line graph of the X and Y coordinate data. LINE is called with the statement

```
CALL LINE(X,Y,N,K)
```

where X and Y are the arrays, N is the number of points to be plotted, and K indicates that every K^{th} value is to be plotted.

The NUMBER Subroutine

NUMBER plots a floating point number in the manner specified in the CALL statement. The calling sequence is

```
CALL NUMBER(X,Y,HEIGHT,FNUM,ANGLE,NDIGIT)
```

where X and Y are the coordinates of the plot position.

FNUM is a floating point constant or variable whose value is the height in inches of the character to be plotted.

ANGLE is the character orientation in degrees counterclockwise from the horizontal.

NDIGIT is the number of decimal places to be plotted. If NDIGIT equals zero, the decimal point is plotted, and if NDIGIT is less than zero, only the integer portion of the number is plotted.

The PLOT Subroutine

The PLOT subroutine moves the plotter pen to the specified position. The calling sequence is

```
CALL PLOT(X,Y,IP)
```

where X and Y are the coordinates of the point to which the pen moves.

IP indicates the pen position while in motion.

IP=2 Pen down; line drawn.

IP=3 Pen up; point drawn.

IP<0 Plots the data in the plot output buffer and resets the origin to the current pen position. The user must call PLOT with a negative IP before terminating execution; otherwise, part of the plot will not be transmitted.

The PLOTF Subroutine

PLOTF initializes the program for off-line plotting. Calling PLOTF instead of PLOTS, described below, causes all the commands that would have been sent to the plotter to be written on a file. The calling sequence is

```
CALL PLOTF(LDEV)
```

where LDEV is the number of a file opened for symbolic output. All plotting commands are written on this file.

The PLOTS Subroutine

The PLOTS subroutine initializes the plotter and must be the first plotting subroutine called. The calling statement is

```
CALL PLOTS(IERR)
```

where IERR is an integer variable which the system sets to a value of 0 if the initialization is successful, and -1 if the plotter is unavailable.

The SCALE Subroutine

The SCALE subroutine computes the initial axis value and the scale factor. The calling sequence is

```
CALL SCALE(X,N,LENGTH,X0,DX)
```

where

- X is the array represented by one axis.
- N is the number of elements in the array X.
- LENGTH is the axis length in inches.
- X0 is the variable name for the computed minimum value of X.
- DX is the variable name for the computed difference in value between two adjacent points on the axis.

The SYMBOL Subroutine

The SYMBOL subroutine permits the user to specify characters to be drawn, the size of the characters, and the angle at which they are drawn. The calling statement is

```
CALL SYMBOL(X,Y,HEIGHT,BCD,ANGLE,NBCD)
```

where

- X and Y are the coordinates, in inches from the plotter origin, of the lower left corner of the symbol to be drawn.
- HEIGHT is the height of the symbol in inches.
- BCD is an integer or real array containing the Hollerith characters to be drawn.
- ANGLE is the character orientation in degrees, measured counterclockwise from the horizontal.
- NBCD is the number of characters from BCD to be drawn.

The WHERE Subroutine

The WHERE subroutine returns the present plotter position. The CALL statement is:

```
CALL WHERE(X,Y)
```

The position is returned as (X,Y).

SECTION 6

BINARY LIBRARY PROGRAM

CARMEL is the TYMCOM-X FORTRAN program for creating and updating subprogram libraries.

To expedite subprogram loading, CARMEL automatically creates a subprogram directory for the entire library and stores it on the library file.

CARMEL uses an ordered list of the subprograms which are to be included in the library and a list of the file containing those subprograms, then selects the subprograms from the appropriate files; it can also list subprograms in an existing library. CARMEL uses only relocatable binary files and assumes the files named have the extension .REL. If a file with a blank extension is to be used, the user types the file name followed by a period.

CREATING OR UPDATING A LIBRARY

CARMEL is called from TYMEX with the command:

-R CARMEL↵

The program prompts for the required input information. Each user response is followed by a Carriage Return. When the user calls CARMEL, the system responds:

OLD LIBRARY *

If the user wishes to create a new library, he names any file containing one or more of the subprograms to be included in the new library. If the user wishes to update an existing library, he types the name of the file to be modified. To use a file from another user's directory, the user types the file name followed by the user's file directory number enclosed in brackets, that is:[1]

file name[file directory number]

The next information requested by the system is:

NEW LIBRARY *

The user types the name of the file on which he wishes to store the new or updated library. The new library file name may be the same as the old library file name.

1 - The user may type PPN in TYMEX to determine his file directory number. This number has the form of two integers separated by a comma.

CARMEL is now ready for the ordered subprogram list; the prompt is

LIBRARY LIST *

after which the user types the name of a file which contains the list of all subprograms to be included in the library. The subprograms must be listed in the order they are to appear in the library. If the user types a Carriage Return in response to this question, the order of the subprograms is the same as in the old library. The file named contains a list of the subprograms to be included. The subprogram names are separated by a character other than a letter or a digit and the list must be terminated with a #. Even though the file named is not a relocatable binary file, the user must indicate a file name extension by typing a period and the actual file name extension or merely a period to indicate a blank extension.

The next system prompt is:

NEW FILE NAME - EXTRA RETURN WHEN DONE
*

The user types the names of all the files which contain one or more subprograms to be added to the old library or to replace subprogram versions on the old library. The user follows each file name with a Carriage Return. When all the files have been named, the user types an extra Carriage Return. CARMEL searches each of the new files and the old library for the subprograms named.

If no new files are required, the user may type a single Carriage Return.

LISTING A LIBRARY

If the user wishes a listing of the subprograms on an existing library file, he types the file name followed by the characters /L or /A in response to the prompt:

OLD LIBRARY *

Instead of the usual prompt requesting the name of the new library, the system prints

OUTPUT TO

and the user responds with a device name followed by a colon (:). Normally, the user will request a listing on the terminal with the response TTY:. If the /L option is chosen, the system lists the names of the subprograms in the old library on the device specified. If the /A option is chosen, the system lists each subprogram name, the entry point of each subprogram, the starting location word number, and the block number on the specified device.

EXAMPLE USING CARMEL

The user has three symbolic files, LIB1, PROD, and QUADR; he wishes to create a single library of the routines written on these files.

```
-EDITOR↵
*APPEND↵                          The user creates a file containing a list of the programs
SUM PRODCT DAFS TCOND1#↵          he wants in the library in the order they are to appear.
*WRITE LIST↵                      He names the file LIST.
NEW FILE↵

25 CHRS
*QUIT↵

-COMPILE LIB1,PROD,DAF↵           The user must compile his symbolic files because CARMEL
F-IV:  LIB1                       works only with relocatable binary files. The COMPILE
 TCOND1                           command automatically creates and saves a relocatable
 SUM                              binary file with the extension .REL.
F-IV:  PROD
 PRODCT
F-IV:  DAF
 DAFS

EXIT

-R CARMEL↵

OLD LIBRARY *LIB1↵                The file named here may be any of the files LIB1, PROD, or QUADR,
                                  since the user is creating a new library rather than modifying an old one.
NEW LIBRARY *LIB2↵

LIBRARY LIST *LIST.↵              The file name is followed by a period to indicate a blank extension.

NEW FILE NAMES - EXTRA RETURN WHEN DONE
*PROD↵          CARMEL searches for the required subprograms on the files
*DAF↵           PROD.REL, QUADR.REL, and the old library file LIB1.REL.
*↵

EXIT

-R CARMEL↵

OLD LIBRARY *LIB2/L↵

OUTPUT TO  TTY:↵                  The user requests the list to be printed on the terminal.
                                  The colon must follow TTY.
SUM
PRODCT
DAFS
TCOND1

EXIT

-
```

APPENDIX

SCIENTIFIC SUBROUTINE PACKAGE

STATISTICAL ROUTINES

ABSNT — Tests missing or zero values for each observation in a matrix.

AUTO — Calculates the autocovariances for lags $0,1,2,\ldots,(L-1)$, given a time series of observations $A_1, A_2, \ldots, A_n$ and a number L.

AVCAL — Performs the calculus for the general k-factor experiment: operator Σ and operator Δ.

AVDAT — Places data for analysis of variance in properly distributed positions of storage (used in conjunction with AVCAL and MEANQ).

BOUND — Selects from a set of observations the number of observations under, between, and over two given bounds for each variable.

CANOR — Performs a canonical correlation analysis between two sets of variables.

CHISQ — Calculates degrees of freedom and chi-square for a given contingency table of observed frequencies.

CORRE — Calculates means, standard deviations, sums of cross-products of deviations from means, and product moment correlation coefficients.

CROSS — Calculates the cross covariances of series B lagging and leading A, given two time series $A_1, A_2, \ldots, A_n$ and $B_1, B_2, \ldots, B_n$, and given a number L.

DISCR — Performs a discriminant analysis.

DMATX — Calculates means of variables in each group and a pooled dispersion matrix for the set of groups in a discriminant analysis.

EXSMO — Calculates a smoothed series given a time series and a smoothing constant.

GAUSS — Computes a normally distributed random number with a given mean and standard deviation.

GDATA	Generates independent variables up to the m^{th} power (the highest degree polynomial specified) and calculates means, standard deviations, sums of cross-products of deviations from means, and product moment correlation coefficients.
KRANK	Computes the Kendall rank correlation coefficient.
LOAD	Calculates the coefficients of each factor by multiplying the elements of each normalized eigenvector by the square root of the corresponding eigenvalue.
MEANQ	Performs the mean square operation for the general k-factor experiment.
MOMEN	Computes four moments for grouped data on equal class intervals.
MULTR	Performs a multiple regression analysis for a dependent variable and a set of independent variables.
NROOT	Finds eigenvalues and eigenvectors of a special nonsymmetric matrix.
ORDER	Constructs from a larger matrix of correlation coefficients a subset matrix of intercorrelations among independent variables with the dependent variable.
QTEST	Determines the Cochran Q-test statistic.
RANDU	Computes uniformly distributed random floating point numbers between 0 and 1.0 and integers in the range 0 to 2^{15}.
RANK	Ranks a vector of values.
SMO	Calculates the smoothed or filtered series, given a time series, a selection integer, and a weighting series.
SRANK	Measures the correlation between two variables by means of the Spearman rank correlation coefficient.
SUBMX	Copies from a larger matrix of observations a subset matrix of those observations which have satisfied certain conditions.
SUBST	Derives a subset vector indicating which observations in a set have satisfied certain conditions of the variable.
TAB1	Tabulates the frequencies and percent frequencies over class intervals for a selected variable in an observation matrix.

TAB2 Performs a two-way classification of the frequency, percent frequency, and other statistics over given class intervals, for two selected variables in an observation matrix.

TALLY Calculates total, mean, standard deviation, minimum, and maximum for each variable in a set of observations.

TIE Calculates the correlation factor due to ties.

TRACE Finds the number of eigenvalues which are greater than or equal to the value of a specified constant.

TTSTT Computes certain t-statistics on the means of populations under various hypotheses.

TWOAV Tests whether a number of samples are from the same population, using the Friedman two-way analysis of variance test.

UTEST Tests whether two independent groups are from the same population by means of the Mann-Whitney U-test.

VARMX Performs orthogonal rotations on an m by k factor matrix in such a way that

$$\sum_j \left\{ m \sum_i \left(a_{ij}^2 / h_i^2 \right)^2 - \left[\sum_i \left(a_{ij}^2 / h_i^2 \right) \right] \right\}^2$$

is a maximum.

WTEST Computes the Kendall coefficient of concordance.

MATRIX SUBROUTINES

ARRAY Converts a data array from single dimension to double dimension or vice versa.

CADD Adds a column of one matrix to a column of another matrix.

CCPY Copies a specified column of a matrix into a vector.

CCUT Partitions a matrix between specified columns to form two resultant matrices.

CINT Interchanges two columns of a matrix.

CSRT	Sorts the columns of a matrix.
CSUM	Adds the elements of each column of a matrix to form a row vector.
CTAB	Tabulates the columns of a matrix to form a summary matrix.
CTIE	Adjoins two matrices with rows of the same length to form one resultant matrix.
DCLA	Sets each diagonal element of a matrix equal to a given scalar.
DCPY	Copies diagonal elements of a matrix into a vector.
EIGEN	Finds eigenvalues and eigenvectors of a real symmetric matrix.
GMADD	Adds two general matrices to form a resultant general matrix.
GMPRD	Multiplies two general matrices to form a resultant general matrix.
GMSUB	Subtracts one general matrix from another to form a resultant matrix.
GMTRA	Transposes a general matrix.
GTPRD	Premultiplies a general matrix by the transposition of another general matrix.
LOC	Computes a vector subscript for an element in a matrix of some specified storage mode.
MADD	Adds two matrices element by element to form a resultant matrix.
MATA	Premultiplies a matrix by its transposition to form a symmetric matrix.
MCPY	Copies an entire matrix.
MFUN	Applies a function to each element of a matrix to form a resultant matrix.
MINV	Inverts a matrix.
MPRD	Multiplies two matrices to form a resultant matrix.
MSTR	Changes the storage mode of a matrix.

MSUB	Subtracts two matrices element by element to form a resultant matrix.
MTRA	Transposes a matrix.
RADD	Adds a row of one matrix to a row of another matrix.
RCPY	Copies a specified row of a matrix into a vector.
RCUT	Partitions a matrix between specified rows to form two resultant matrices.
RECP	Calculates the reciprocal of an element of a matrix.
RINT	Interchanges two rows of a matrix.
RSRT	Sorts the rows of a matrix.
RSUM	Adds the elements of each row of a matrix to form a column vector.
RTAB	Tabulates the rows of a matrix to form a summary matrix.
RTIE	Adjoins two matrices with columns of the same length to form one resultant matrix.
SADD	Adds a scalar to each element of a matrix to form a resultant matrix.
SCLA	Sets each element of a matrix equal to a given scalar.
SCMA	Multiplies a column of a matrix by a scalar and adds the result to another column of the same matrix.
SDIV	Divides each element of a matrix by a scalar to form a resultant matrix.
SIMQ	Solves a set of simultaneous linear equations.
SMPY	Multiplies each element of a matrix by a scalar to form a resultant matrix.
SRMA	Multiplies a row of a matrix by a scalar and adds the result to another row of the same matrix.
SSUB	Subtracts a scalar from each element of a matrix to form a resultant matrix.

TPRD	Transposes a matrix, then multiplies it by another to form a resultant matrix.
XCPY	Copies a part of a matrix.

MATHEMATICAL SUBROUTINES

BESI	Computes the I Bessel function for a given argument and order.
BESJ	Computes the J Bessel function for a given argument and order.
BESK	Computes the K Bessel function for a given argument and order.
BESY	Computes the Y Bessel function for a given argument and order.
CEL1	Computes the complete elliptic integral of the first kind.
CEL2	Computes the generalized complete elliptic integral of the second kind.
CS	Computes the Fresnel integrals for a given value of the argument.
EXPI	Computes the exponential integral in the range from -4 to infinity.
FORIF	Produces the Fourier coefficients for a given periodic function.
FORIT	Produces the Fourier coefficients of a tabulated function.
GAMMA	Computes the value of the gamma function for a given argument.
LEP	Computes the values of the Legendre polynomials P(N,X) for argument value X and orders 0 to N.
PADD	Adds two polynomials.
PADDM	Adds coefficients of one polynomial to the product of a factor by coefficients of another polynomial.
PCLA	Replaces one polynomial with another.
PCLD	Shifts the origin of a coefficient vector of a polynomial.
PDER	Finds the derivative of a polynomial.

PDIV	Divides one polynomial by another.
PGCD	Determines the greatest common division of two polynomials.
PILD	Evaluates a polynomial and its first derivative for a given argument.
PINT	Finds the integral of a polynomial with the constant of integration equal to zero.
PMPY	Multiplies two polynomials.
PNORM	Normalizes the coefficient vector of a polynomial.
POLRT	Computes the real and complex roots of a real polynomial.
PQSD	Performs quadratic synthetic division.
PSUB	Subtracts one polynomial from another.
PVAL	Evaluates a polynomial for a given value of the variable.
PVSUB	Substitutes the variable of one polynomial with another polynomial.
QATR	Finds the integral of a tabulated function by quadrature.
QSF	Integrates an equidistantly tabulated function using Simpson's rule.
RKGS	Solves a system of first-order ordinary differential equations given initial values, using the Runge-Kutta method.
RK1	Integrates a given function using the Runge-Kutta technique and produces the final computed value of the integral.
RK2	Integrates a given function using the Runge-Kutta technique and produces tabulated values of the computed integral.
RTMI	Determines a root of the general nonlinear equation f(X)=0, using Mueller's iteration scheme.
RTNI	Refines the initial guess x_0 of a root of the general nonlinear equation f(X)=0. Newton's iteration scheme is used.
RTWI	Refines the initial guess x_0 of a root of the general nonlinear equation x=f(X). Wegstein's iteration scheme is used.
SICI	Computes the sine and cosine integrals.